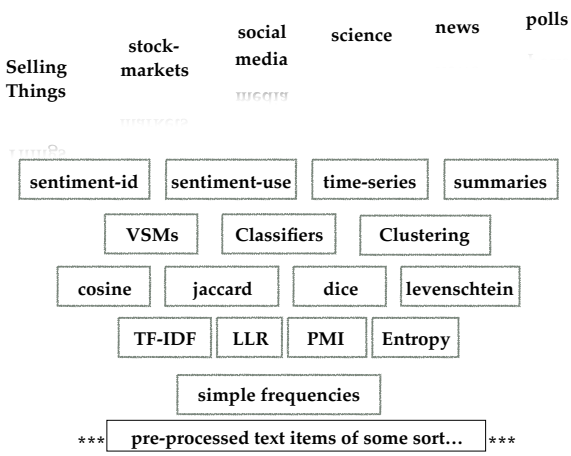
De Data:

The One Where We Do De Data

Lecture 2: Text Analytics for Big Data
Mark Keane, Insight/CSI, UCD



The Basic View

- Well, the course is called Text Analytics
- One expects the *data* to be text, words, clauses, sentences, paragraphs, and other things written in ***natural language***
- Here, we consider this data and ***pre-processing*** "raw" text for future use

What's the Problem?

"It was in the winter of '69 that we encountered the first sign that the U.S.A. was entering into a conflict, streked (sic) by the GOP, against some unspecified Evil. Mother feared the worst for future fish catches. The Coalition of Baddies was afraid of war and fishing immediately ceased in the South China Sea."

What's the Problem?

"It was in the winter of '69 that we encountered the first sign that the U.S.A. was entering into a conflict, streked (sic) by the GOP, against some unspecified Evil. Mother feared the worst for future fish catches. The Coalition of Baddies was afraid of war and fishing immediately ceased in the South China Sea."

What are these ?

What's the Problem?

"It was in the winter of '69 that we encountered the first sign that the U.S.A. was entering into a conflict, streked (sic) by the GOP, against some unspecified Evil. Mother feared the worst for future fish catches. The Coalition of Baddies was afraid of war and fishing immediately ceased in the South China Sea."

This is really "stoked"

What's the Problem?

"It was in the winter of '69 that we encountered the first sign that the U.S.A. was entering into a conflict, streked (sic) by the GOP, against some unspecified Evil. Mother feared the worst for future fish catches. The Coalition of Baddies was afraid of war and fishing immediately ceased in the South China Sea."

"feared" & "afraid" are sort of the same !

What's the Problem?

"It was in the winter of '69 that we encountered the first sign that the U.S.A. was entering into a conflict, streked (sic) by the GOP, against some unspecified Evil. Mother feared the worst for future fish catches. The Coalition of Baddies was afraid of war and fishing immediately ceased in the South China Sea."

"fish" & "fishing" look the same but are different !

Not all words are equal…

"winter sign U.S.A. conflict GOP Evil Mother feared fish catches Coalition Baddies afraid war fishing ceased South China Sea"

find the content-bearing words

Will end with…

"winter encounter sign usa enter conflict stok gop region east asia mother fear future fish catch coalition bad fear afraid fish cease south china sea."

Pre-processing does this; all words are modified, stemmed, dropped and the output is the raw data ("words") used in the analysis

Overview

- Basics of Natural Language Processing (NLP)
- Pre-processing to modify text:
  - tokenisation, stemming, POS tagging, lemmatisation, fixing spellings
- Pre-processing to exclude (some) text
  - removing stop words
- When pre-processing helps or not ?

Important Point

- Pre-processing is not just about taking things out; stripping off stems, removing stops etc…
- It may also be about putting things in; like POS tags, syntax, entity tags, lexical chains

NLP 101

I'll -ology you

- NLP is about meaning, parsing stops short:
  - *Phonology & Morphology*: sounds & words
  - *Syntax*: traditionally syntactic constituents
  - *Meaning*: Semantics & Pragmatics

I'll -ology you.

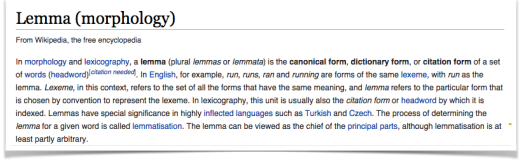
- Natural Language Processing concerns itself with recovering meaning from spoken sounds or written strings, usually using different levels of analysis:
  - Phonology (speech alone)
  - Morphology (speech & writing)
  - Syntax (speech & writing)
  - Semantics (speech & writing)
  - Pragmatics (speech & writing)

I'll -ology you..

- *Phonology*: analysis of sounds into phonemes, sets of which form words
- *Morphology*: analysis of sounds/signs into morphemes (smallest grammatical unit of a language) not= word, morphemes may or may not stand alone ("dog" & "s" versus "dog" & "dogs")
- *Syntax*: analysis of rules of combination of grammatical units of a language to form sentences

I'll -ology you…

- *Semantics*: analysis of how the signs of a language come to convey its meaning; the study of meaning at the levels of words, phrases, sentences, and larger units of discourse (termed texts or narratives)
- *Pragmatics:* analysis of how context contributes to meaning (*yeah…yeah*)



Lemma (morphology)

From Wikipedia, the free encyclopedia

In morphology and lexicography, a **lemma** (plural *lemmas* or *lemmata*) is the **canonical form**, **dictionary form**, or **citation form** of a set of words (headword)[citation needed]. In English, for example, *run*, *runs*, *ran* and *running* are forms of the same lexeme, with *run* as the lemma. *Lexeme*, in this context, refers to the set of all the forms that have the same meaning, and *lemma* refers to the particular form that is chosen by convention to represent the lexeme. In lexicography, this unit is usually also the *citation form* or headword by which it is indexed. Lemmas have special significance in highly inflected languages such as Turkish and Czech. The process of determining the *lemma* for a given word is called lemmatisation. The lemma can be viewed as the chief of the principal parts, although lemmatisation is at least partly arbitrary.

- Is on aspects of morphology and syntax to find…
- …parts of speech for words (fish is a noun, fishing is a verb, fishy is an adjective)
- …roots and/or lemmas of words (the root fish~ from fishing, fishes, fished; the root be~ from am, are, is)

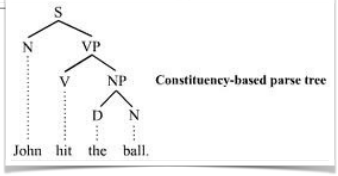
http://en.wikipedia.org/wiki/Lemma_(morphology)

Focus on Parsing…

Parse - Merriam-Webster Online
www.merriam-webster.com/dictionary/**parse**
grammar : to divide (a sentence) into grammatical parts and identify the parts and their relations to each other. : to study (something) by looking at its parts ...

Parsing
Programming Language

Parsing or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar. The term parsing comes from Latin pars, meaning part. Wikipedia



Constituency-based parse tree

Text Pre-Processing

Pre-Processing: De Guts

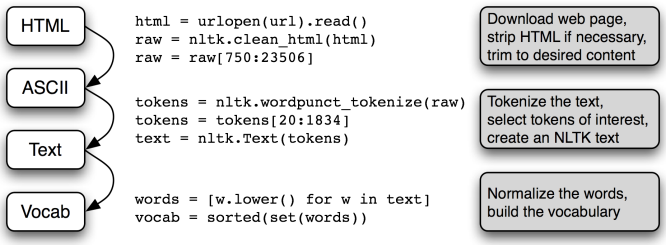
Our Focus…

- Text pre-processing is the poor-farmer cousin of full NLP; its not really about meaning
- Its about cleaning up text-data for future use
- Uses ideas from NLP (eg syntactic analysis, parsing) … but is not often full NLP
- Ultimately, it seldom recovers meaning

Standard Tasks

- *Tokenisation & Normalisation*: finding boundaries between word-like entities in character string
- *Fixing Misspellings*: where possible
- *Stemming, lemmatisation, POS-tagging*: finding slightly deeper identities between words (fished, fishing)
- *Removing Stop Words:* maximising the content-full words in the document/corpus
- *Entity Extraction:* identifying conceptual entities behind words

Typical Python Pipeline



But…First Remember

- *Document encodings*: byte sequence turned into a character sequence, ok if ASCII but are different encoding schemes (Unicode UTF-8); need to identify (meta-data) and decode
- *Document formats*: DOC, PDF, XML, HTML may all need own converters to ensure characters are identified
- *Document text*: text-part of particular doc needs to be extracted (e.g., from XML, often not clean for sites)

Text Pre-Processing

Tokenisation & Normalisation

What is the Problem ?

- We have some sequence of characters, we need to break into word-like tokens/entities
- No problemo, marko, you pick those things with space-chars around them or fullstops before them…
- Right?

Why Tokenise?

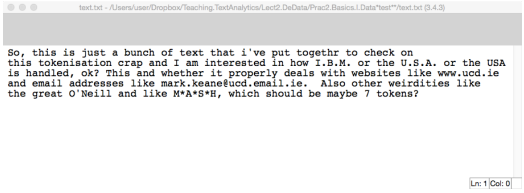
- Well, what about:

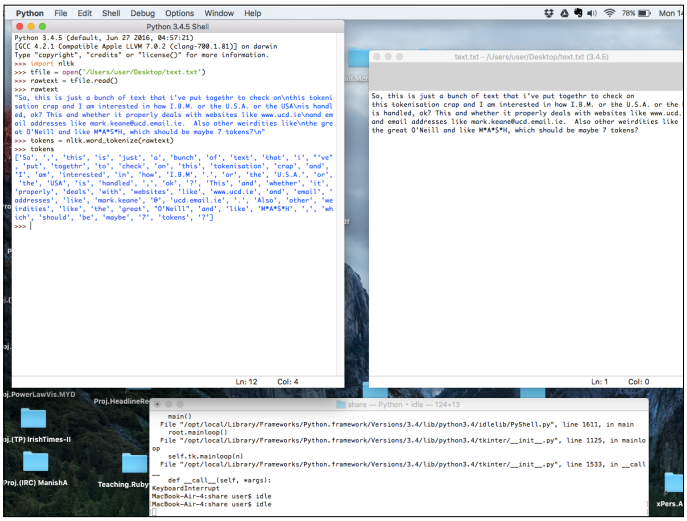
I've:	'I' 've' / 'I've' / Ive
U.S.A. :	'U' 'S' 'A' / 'USA' / 'U.S.A.'
Jonny O'Neill:	'Jonny' 'O' 'Neill' / 'Jonny' 'O'Neill'
www.ucd.ie:	'www' 'ucd' 'ie' / 'www.ucd.ie' / 'www' 'ucd.ie'
mark.keane@ucd.ie:	'mark' 'keane' '@' 'ucd' 'ie' / what?

- Which one you pick has impact on the text data

Tokenising I

- Tokenisers work off general rules and a lot of really quite specific ones too
- Consider the following text file:





Tokenising III

- So, the **nltk** Python tokeniser obviously works off a set of rules for handling things of different types
- Some rules are quite general (eg handling fullstops)
- Other rules are very specific (M*A*S*H)
- **nltk** makes it easy for you to extend these rules; using the **re** package (for regular expressions)

Some Downsides…

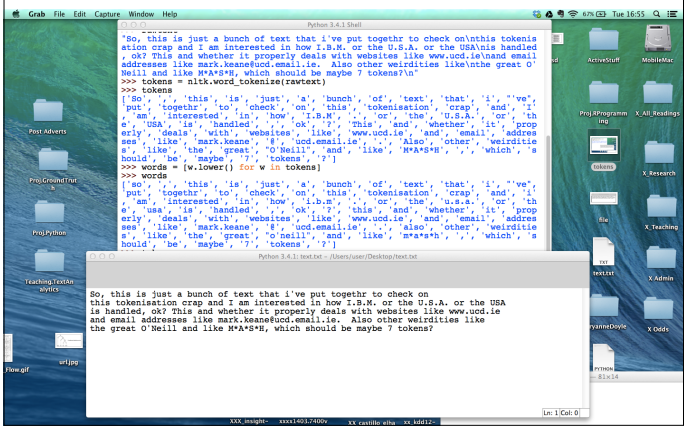
- There are other problems to handle in English, like hyphenation, foreign phrases (*au fait*), numbers, acronyms, etc…
- Problem: solution is quite language-dependent, may need different solutions in:
  - German: where they compound a lot
  - Chinese/Korean: where clear word boundaries are not shown by spaces

Why Normalise ?

- Is often done to reduce of minor differences between tokens, like:
  - accented differences (cliche, cliché)
  - capitals (the, The)
  - acronyms (U.S.A., USA)
- Though there may be more complex solutions involving *equivalence classes*

http://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html

nltk Normalisation = downcase



Text Pre-Processing

Misspelling

Why Fix Spellings ?

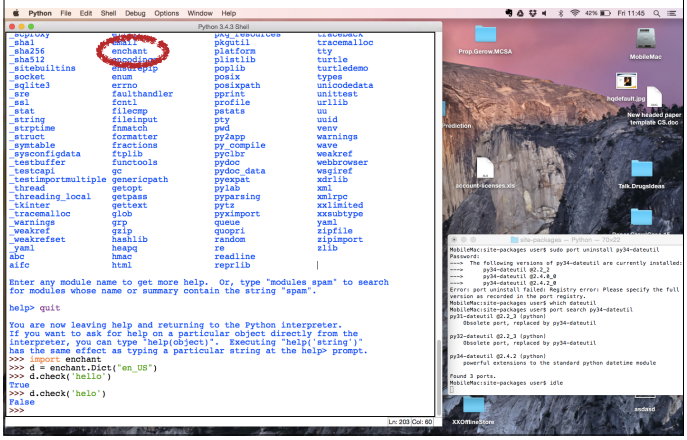
- Obviously, we cannot assume all the words are spelt correctly; books best, tweets worst
- So, after tokenizing and normalisation, one could run a spelling checker over the tokens
- Standard solution is to take a dictionary, compare each word against it, if no entry found, use shortest edit distance to a word in dict

How to Fix Spellings

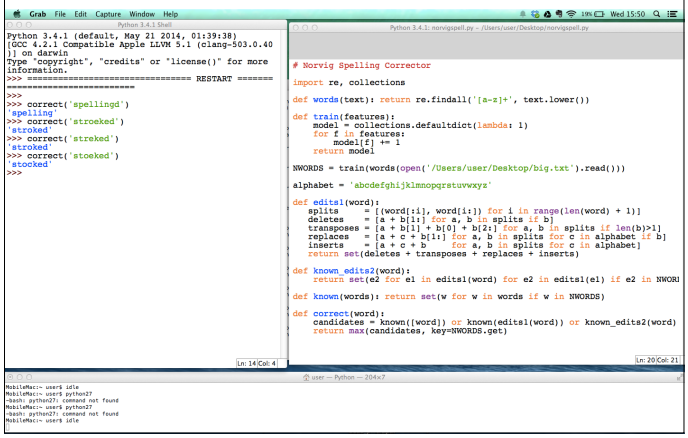
- Python has a number of different packages that do spelling correction and front-ends that will highlight misspellings in text
- the **enchant** package is one of most popular
- So, Mac Install Enchant (may be in spyder)

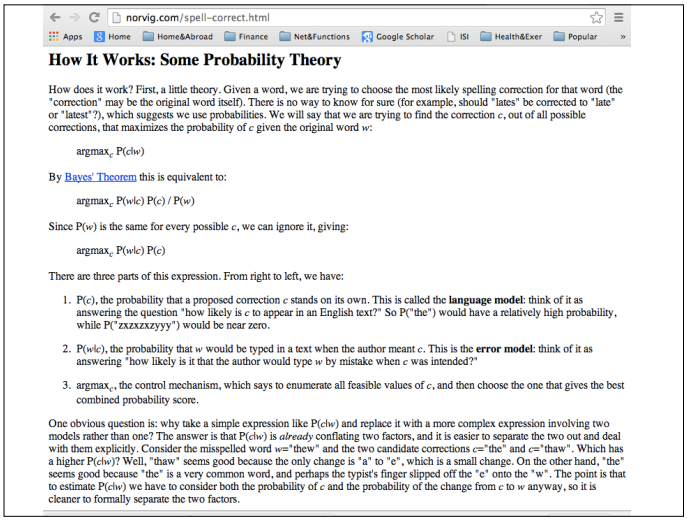
  $ sudo port install py34-enchant

Spellings: **enchant module**



Fixing Spellings: BigData Way





norvig.com/spell-correct.html

How It Works: Some Probability Theory

How does it work? First, a little theory. Given a word, we are trying to choose the most likely spelling correction for that word (the "correction" may be the original word itself). There is no way to know for sure (for example, should "lates" be corrected to "late" or "latest"?), which suggests we use probabilities. We will say that we are trying to find the correction c, out of all possible corrections, that maximizes the probability of c given the original word w:

$$\text{argmax}_c \; P(c|w)$$

By Bayes' Theorem this is equivalent to:

$$\text{argmax}_c \; P(w|c)\, P(c) \,/\, P(w)$$

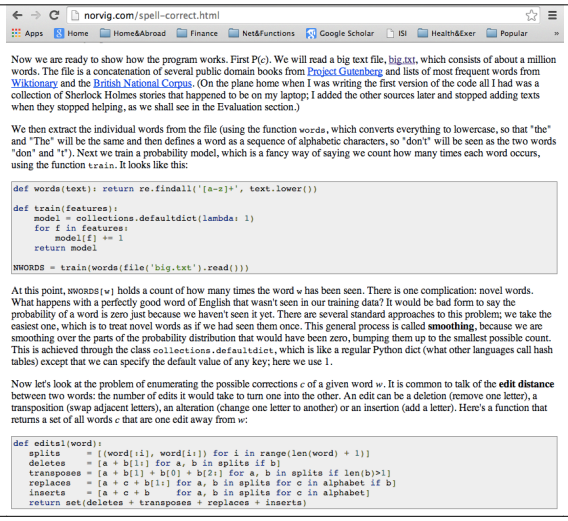
Since $P(w)$ is the same for every possible c, we can ignore it, giving:

$$\text{argmax}_c \; P(w|c)\, P(c)$$

There are three parts of this expression. From right to left, we have:

1. $P(c)$, the probability that a proposed correction c stands on its own. This is called the **language model**: think of it as answering the question "how likely is c to appear in an English text?" So P("the") would have a relatively high probability, while P("zxzxzxzyyy") would be near zero.
2. $P(w|c)$, the probability that w would be typed in a text when the author meant c. This is the **error model**: think of it as answering "how likely is it that the author would type w by mistake when c was intended?"
3. argmax_c, the control mechanism, which says to enumerate all feasible values of c, and then choose the one that gives the best combined probability score.

One obvious question is: why take a simple expression like $P(c|w)$ and replace it with a more complex expression involving two models rather than one? The answer is that $P(c|w)$ is *already* conflating two factors, and it is easier to separate the two out and deal with them explicitly. Consider the misspelled word w="thew" and the two candidate corrections c="the" and c="thaw". Which has a higher $P(c|w)$? Well, "thaw" seems good because the only change is "a" to "e", which is a small change. On the other hand, "the" seems good because "the" is a very common word, and perhaps the typist's finger slipped off the "e" onto the "w". The point is that to estimate $P(c|w)$ we have to consider both the probability of c and the probability of the change from c to w anyway, so it is cleaner to formally separate the two factors.



norvig.com/spell-correct.html

Now we are ready to show how the program works. First $P(c)$. We will read a big text file, big.txt, which consists of about a million words. The file is a concatenation of several public domain books from Project Gutenberg and lists of most frequent words from Wiktionary and the British National Corpus. (On the plane home when I was writing the first version of the code all I had was a collection of Sherlock Holmes stories that happened to be on my laptop; I added the other sources later and stopped adding texts when they stopped helping, as we shall see in the Evaluation section.)

We then extract the individual words from the file (using the function `words`, which converts everything to lowercase, so that "the" and "The" will be the same and then defines a word as a sequence of alphabetic characters, so "don't" will be seen as the two words "don" and "t"). Next we train a probability model, which is a fancy way of saying we count how many times each word occurs, using the function `train`. It looks like this:

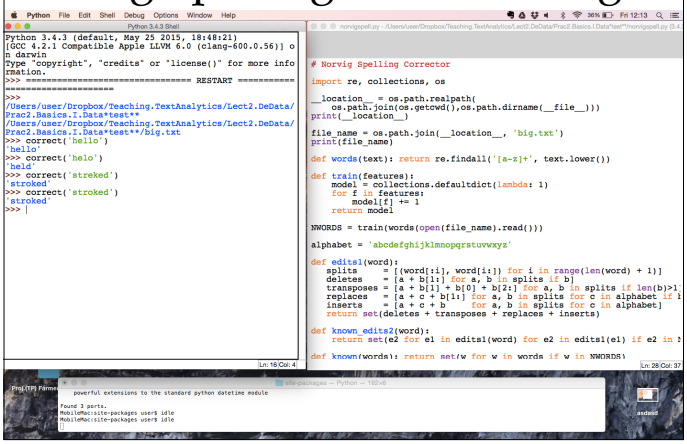
```
def words(text): return re.findall('[a-z]+', text.lower())

def train(features):
    model = collections.defaultdict(lambda: 1)
    for f in features:
        model[f] += 1
    return model

NWORDS = train(words(file('big.txt').read()))
```

At this point, `NWORDS[w]` holds a count of how many times the word `w` has been seen. There is one complication: novel words. What happens with a perfectly good word of English that wasn't seen in our training data? It would be bad form to say the probability of a word is zero just because we haven't seen it yet. There are several standard approaches to this problem; we take the easiest one, which is to treat novel words as if we had seen them once. This general process is called **smoothing**, because we are smoothing over the parts of the probability distribution that would have been zero, bumping them up to the smallest possible count. This is achieved through the class `collections.defaultdict`, which is like a regular Python dict (what other languages call hash tables) except that we can specify the default value of any key; here we use 1.

Now let's look at the problem of enumerating the possible corrections c of a given word w. It is common to talk of the **edit distance** between two words: the number of edits it would take to turn one into the other. An edit can be a deletion (remove one letter), a transposition (swap adjacent letters), an alteration (change one letter to another) or an insertion (add a letter). Here's a function that returns a set of all words c that are one edit away from w:

```
def edits1(word):
   splits     = [(word[:i], word[i:]) for i in range(len(word) + 1)]
   deletes    = [a + b[1:] for a, b in splits if b]
   transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b)>1]
   replaces   = [a + c + b[1:] for a, b in splits for c in alphabet if b]
   inserts    = [a + c + b     for a, b in splits for c in alphabet]
   return set(deletes + transposes + replaces + inserts)
```

Fixing Spellings: Actual Prog



Fixing Spellings: Google Way

- http://www.internetmarketingninjas.com/blog/search-engine-optimization/google-spell-check/

Text Pre-Processing

Stemming & Lemmatising

What is the Problem?

- We have a number of normalised tokens that, we hope, are the words in the document
- But, we are going to be counting them and doing other manipulations on frequency
- So, it is very important that words that are more or less the same are identified…

Why Stem?

- So, we need to recognise variations of the same *stem* or *root**: fish~ for fishes, fishing, fished…
- We also need to recognise when two words are the same but from different syntactic categories (or parts of speech, POS); fish the *noun* and fish the *verb*
- Note, the root form of a word may be quite different **be~** for is, are, am

* may be used differently

Why Stem?

- So, if we convert the variations into the root form then we have a more accurate count of the word
- So, if we distinguish two words that appear the same but are actually different fish-v and fish-n (stemming doesn't really do this…)
- Generally, stemming refers to the recovery of the root forms of words to show these commonalities and differences

Stemming Definition…

Stemming

From Wikipedia, the free encyclopedia

This article **needs attention from an expert on the subject**. Please add a *reason* or a *talk* parameter to this template to explain the issue with the article. Consider associating this request with a WikiProject. *(October 2010)*

For the skiing technique, see Stem (skiing).

In linguistic morphology and information retrieval, **stemming** is the process for reducing inflected (or sometimes derived) words to their stem, base or root form—generally a written word form. The stem need not be identical to the morphological root of the word; it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid root. Algorithms for stemming have been studied in computer science since the 1960s. Many search engines treat words with the same stem as synonyms as a kind of query expansion, a process called **conflation**.

Stemming programs are commonly referred to as **stemming algorithms** or **stemmers**.

Stemming Algorithms

- Technically, stemming is described the removal of morphological affixes
- Porter Stemmer, Snowball stemmers, most used but many more in **nltk** (types and languages)
- http://www.nltk.org/howto/stem.html

affix

verb
3rd person present: **affixes**
/ə'fɪks/

1. stick, attach, or fasten (something) to something else.
   "panels to which he affixes copies of fine old prints"
   synonyms: attach, stick, fasten, bind, fix, post, secure, join, connect, couple; More

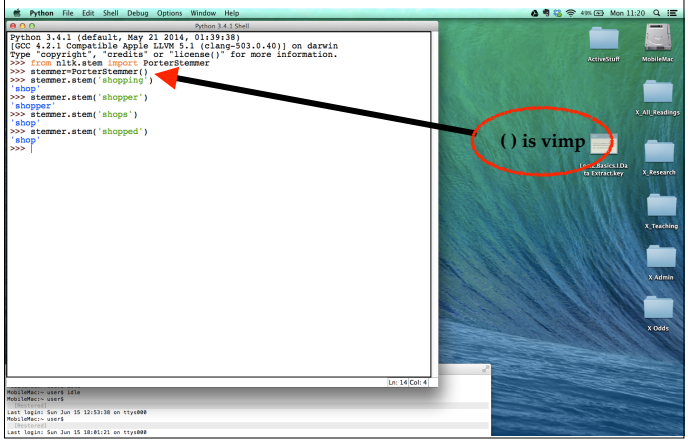
noun GRAMMAR
plural noun: **affixes**
/ˈafɪks/

1. an addition to the base form or stem of a word in order to modify its meaning or create a new word.

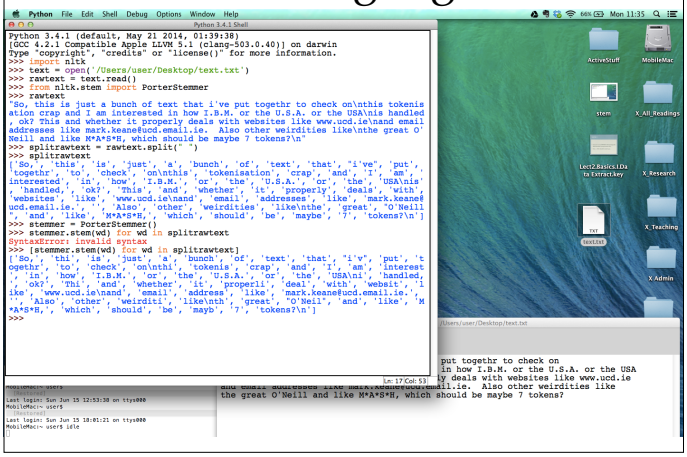
Porter Stemming: Brutal

- Porter Stemming is quite brutal, tho' most of it makes sense (e.g., does plurals well)
- It works off a mixture of specific rules and very general ones (e.g., cutting '-ed' off words)
- http://tartarus.org/martin/PorterStemmer/

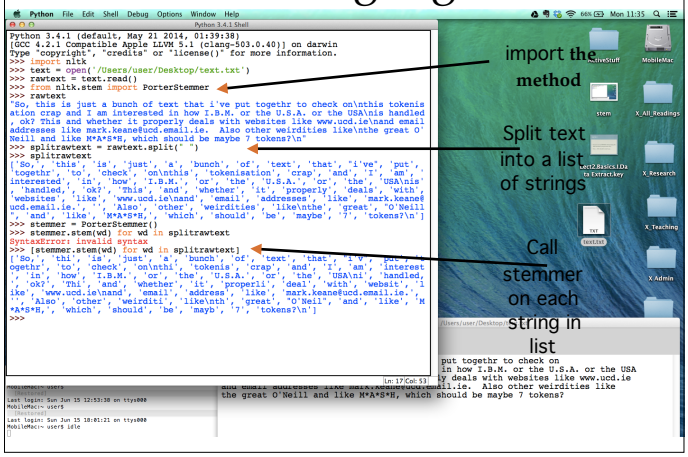
Porter Stemming: Eg



Porter Stemming: Eg



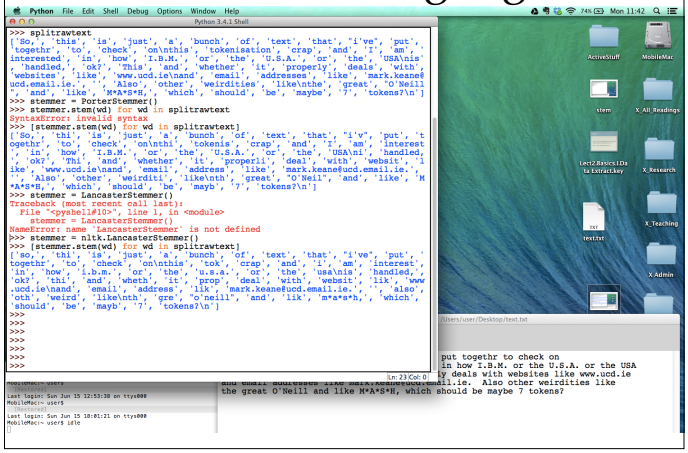
Porter Stemming: Eg



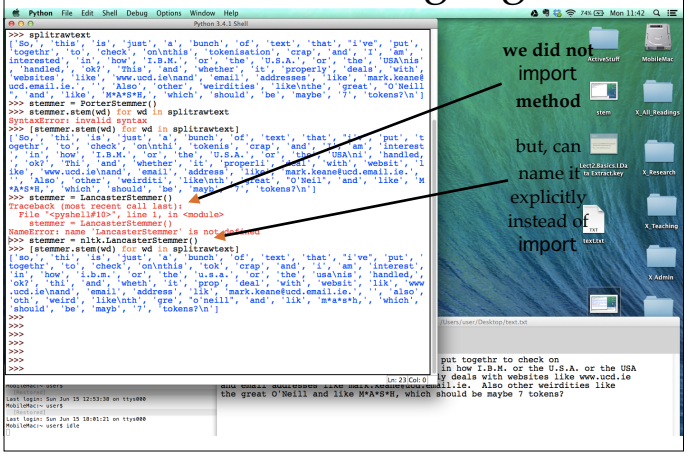
Porter Stemming: Steps

- Step 1: Gets rid of plurals and -ed or -ing suffixes
- Step 2: Turns terminal y to i when there is another vowel in the stem
- Step 3: Maps double suffixes to single ones: | -ization, -ational
- Step 4: Deals with suffixes, -full, -ness etc.
- Step 5: Takes off -ant, -ence, etc.
- Step 6: Removes a final -e

Lancaster Stemming: Eg



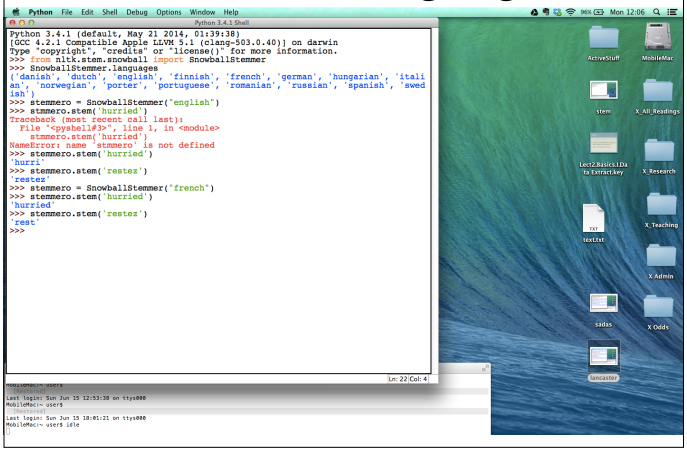
Lancaster Stemming: Eg



Snowball Stemming

- **nltk** has a bunch of off-the-shelf stemmers
- Snowball Stemmer, in **nltk**, is really a language for building stemmers, as they are language-specific this is useful
- Also, you can use it to explicitly switch languages

Snowball Stemming: Eg



Butt Stemmers…But

- Are rather crude; no word meaning disambiguation (bats, batting)
- No POS disambiguation (fish and fish)
- Can't handle irregulars (am, is, are)
- So, sometimes you need more…

www.eecis.udel.edu/~trnka/CISC889-11S/lectures/dan-porters.pdf