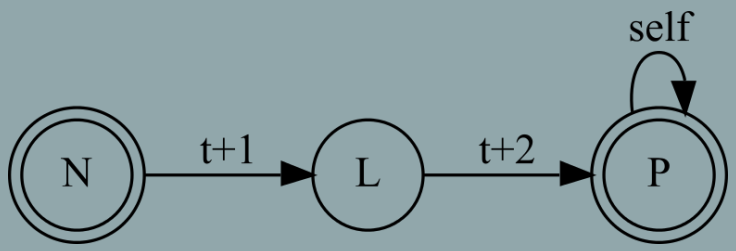
LING-362

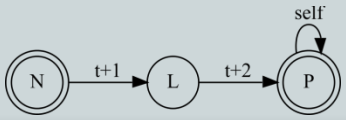
Introduction to **N**atural **L**anguage **P**rocessing

Python & NLTK basics



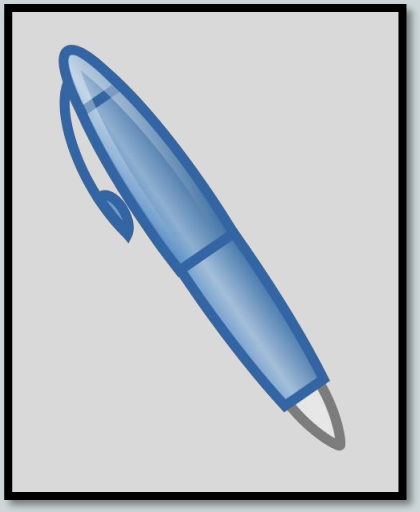
Talks etc.

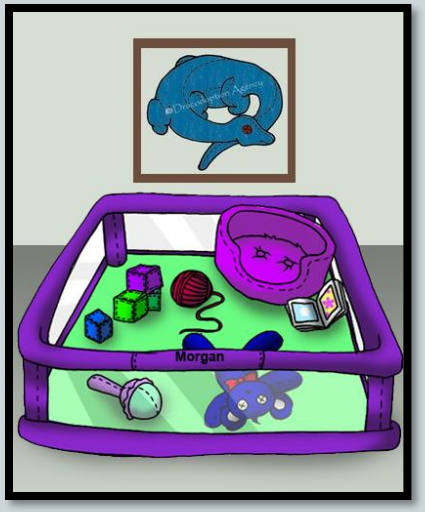
- Sep. 10 – me & co. – Digital Classicist Seminar, Named Entities in Coptic Antiquity
- Sep. 24:
  - Gemma Boleda (UPF Barcelona), Computational Semantics, GU Linguistics Speaker Series
  - 19th Semi-Annual GU-CS Graduate Research – 3/4 talks with CL/IR topics: https://www.georgetown.edu/event/the-19th-semi-annual-gu-cs-graduate-research-presentation-days-session-i/

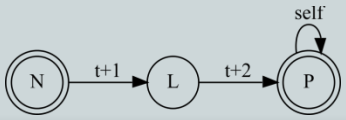


Discussion – Bar Hillel's pen

- Is Bar Hillel right?
- Can context ever get ***pen*** right?
  - Does perfect MT require perfect AI?
  - Can we get much better even without it?

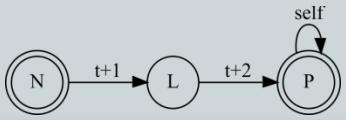






How right is Bar Hillel today?

✗	The box is in the pen	La caja está en el bolígrafo
✗	The box is in the pen	Die Kiste ist im Stift
✗	The box is in the pen	箱はペンの中にあります
✗	The box is in the pen	La boîte est dans le stylo
✗	The box is in the pen	Pudełko jest w długopisie
✗	The box is in the pen	盒子在笔里



How right was Bar Hillel last year?

✗ The box is in the pen × La caja está en la pluma

✗ The box is in the pen × Die Box befindet sich im Stift

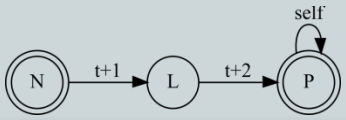
✗ The box is in the pen × 箱はペンに入っています

✗ The box is in the pen × La boîte est dans le stylo

✗ The box is in the pen × Pudełko znajduje się w długopisie

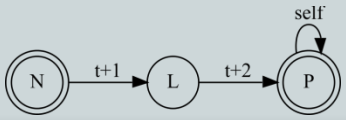
Three(!) years ago:

✓ The box is in the pen. × La caja está en el corral.



Today

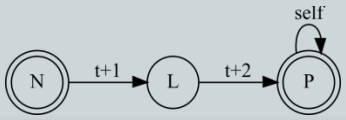
- Python basics
  - Doing math
  - Dealing with variables
  - Processing strings of text
  - Starting our very first program
- NLTK
  - Our first Class of objects
  - Some neat ready-made methods



Python basics

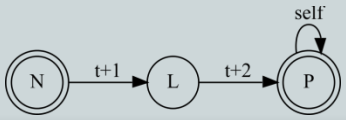
- Programming is all about computing:
  - Getting some values from somewhere
  - Storing them in variables
  - Doing some calculations
  - Outputting the result

- First example: doing math



Starting the Python console

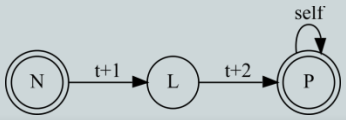
- Python is an 'interpreted' language
  - Commands are read one at a time (from script / CLI)
  - No **compilation**
  - Slower than (some) compiled languages
  - Easier to modify / debug
  - Cross platform (only interpreter OS-specific)
- You can start the interpreter from the command line (CLI, terminal) by running ***python*** ***(*or python3 etc.)***



Time to ask Python some questions

- Assigning from variables:

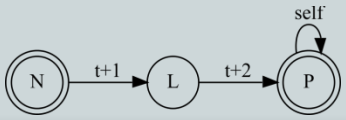
```
>>> b = 8 * a
>>> b
24
>>> (a + a) / 4 # Normal – in Python 3
1.5
>>> (a + a) // 4 # What's this?
1
>>> (a + a) % 4 # The modulo '%' gives us the remainder
2
```



Time to ask Python some questions

- Basic math:

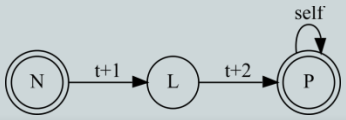
```
>>> 5 * 8
40
>>> a = 3 # assign the variable 'a' the value 3
>>> a * a
9
>>> a ** 2 # use ** for powers
9
```



A word about data types

- Python treats numbers like 3, 4 (and our **a**) as **integers**
- Calculations with integers normally result in integers (no fractions)
- We can turn a variable into a **float** explicitly

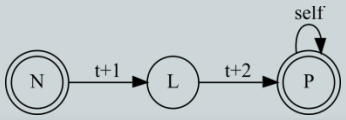
```
>>> a = 3
>>> a = float(a)
>>> a
3.0
>>> (a + a) // 4
1.5
>>> (a + a) / 4 # This now works in Python 2 and 3
1.5
```



A word about data types

- We can convert integers to strings:

```
>>> a = 4
>>> a
4
>>> b = str(a)
>>> b
'4'
```



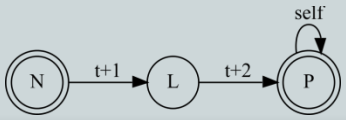
A word about data types

- And back:

```
>>> c = int(b)
>>> c
4
```

- Note the error – be aware of your data types!

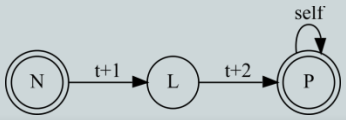
```
>>> c+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```



Dealing with text

- The most frequent data we'll deal with is **characters**, or more often **strings** of characters
- Two strings can use ‘math operations’ too:

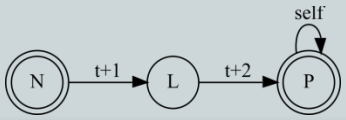
```
>>> word1 = 'hello'
>>> word2 = 'world'
>>> word1 + ' ' + word2 # Adding strings = concatenating
'hello world'
>>> word1 * 2 # Multiplying = repeating
'hellohello'
```



Breaking down strings

- Strings are really just lists of characters
- We can access individual positions, starting with 0:

```
>>> word1[0]        # Give me character 0 – the first one
'h'
>>> word1[0:2]      # Start at 0 – stop just before 2 (first L)
'he'
>>> word1[3:]       # Start at 3, don't stop
'lo'
>>> word1[-1]       # Give me the -1th character = the last
'o'
>>> word1[0:-1]     # Start at 0 – stop just before -1 (=the last)
'hell'
```



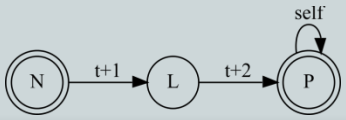
Using built-in functions

- Python comes with many useful **functions**
- Called by name with brackets and **arguments**
- For example, **max()** gives you the largest of the input numbers:

```
>>> max(3,6)
6
```

- The function **len()** gives you the length of a variable:

```
>>> len('hello')
5
```

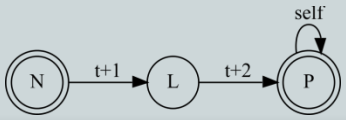


Boolean comparisons

- Python can also tell you if something is true:

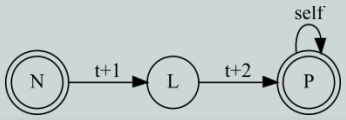
```
>>> 5 > 2
True
>>> 7 <= 6
False
>>> a = 6
>>> a == 7
False
>>> a != 3
True
```

The logical values **True** and **False** are also called Booleans (after George Boole)



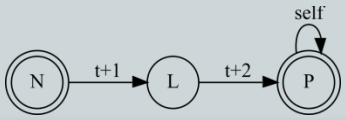
Exercise

- We will write (toy) code to create the Classical Greek perfect from present tense forms:
  - *luo* "I release" *thuo* "I sacrifice"
  - *leluka* "I have released" *tethuka* "I've sacrificed"
- Tips:
  - **Input** – present form – need to get rid of the 'o'
    - word_without_o = … ?
  - **Output** –
    - need to duplicate first consonant: first_consonant = ..?
    - add -ka suffix: **print**(x + y + "ka")



Solution

```
word = "luo"
first_consonant = word[0]  # this will be "l"
word_without_o = word[0:-1]  # this will be "lu"
# print leluka:
print(first_consonant + "e" + word_without_o + "ka")
```

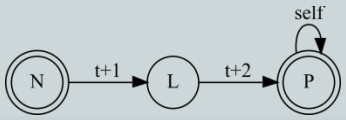


Testing this in the IDE

- Write your code in PyCharm
- Click on a line number to set a breakpoint
- Use: run > debug

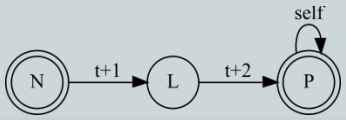
```
1    word = "luo"  word: 'luo'
2    first_consonant = word[0]  first_consonant: 'l'
3    word_without_o = word[0:-1]  word_without_o: 'lu'
4    print(first_consonant + "e" + word_without_o + "ka")  # leluka
5
```

- (This script is also available in Canvas > Files > Code)



Doing Linguistics with Python

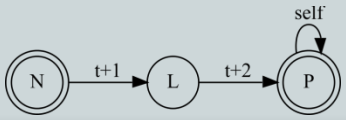
- You can do lots of string editing yourself
- Write your own code for linguistic tasks
- But there are MANY libraries out there to do things for you:
  - NLTK – the Natural Language Toolkit
  - spaCy (https://spacy.io/)
  - Stanza (https://stanfordnlp.github.io/stanza/)
  - Trankit (https://github.com/nlp-uoregon/trankit)
  - …



Doing Linguistics with Python

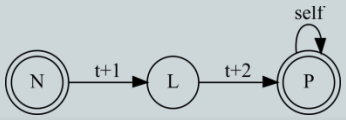
- Using things off the shelf is generally **a good idea**
  - Other people will know what you used
  - Code easier to maintain
    (Object Oriented Programming – more later)
  - Less work for you

- There are also some cons:
  - Might not do exactly what you expect
  - Bugs harder to trace
  - Version incompatibilities
  - Security vulnerabilities



NLTK

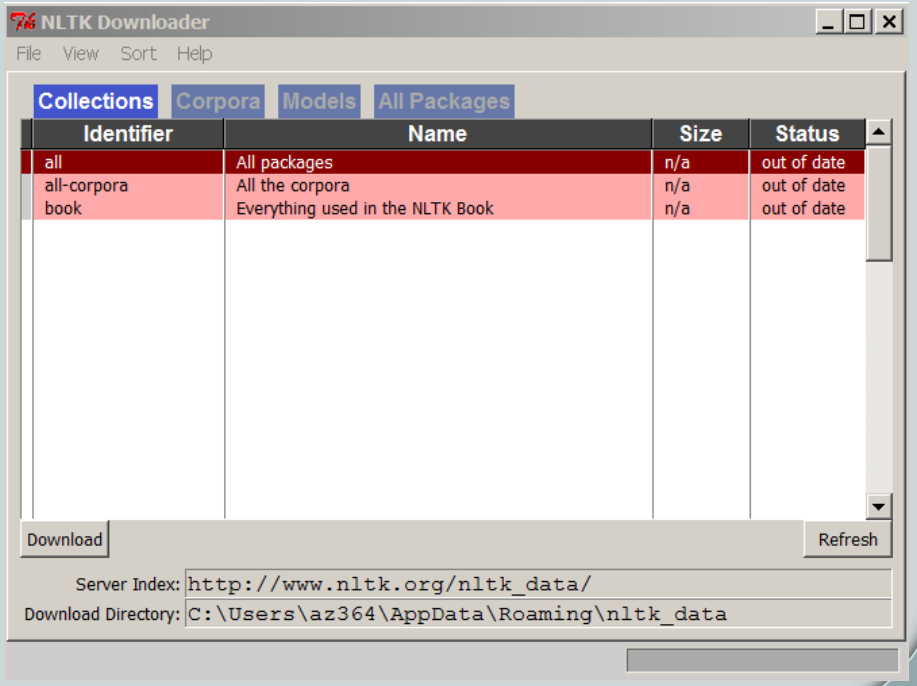
- Basically a collection of teaching demo-type tools by Steven Bird and colleagues
- Not recommended by some for large scale applications (alternatives: e.g. *spaCy*, *Stanza*, neural network libraries like *flair*, …)
- But actually widely used in a lot of applications, especially if speed is not crucial

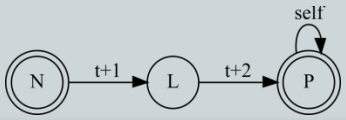


NLTK – a quick taste

- Download and install from
  http://www.nltk.org
- Once installed, run Python in terminal
- Download resources:

```
>>> import nltk
>>> nltk.download()
```

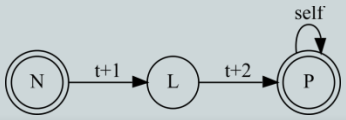




NLTK – a quick taste

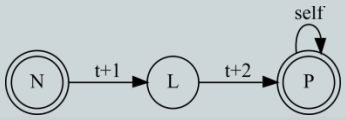
- With the resources installed, we can play with some texts:

```
>>> from nltk.book import *
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```



NLTK – a quick taste

- What are these texts?
  - Objects of the type or **Class *Text***
  - A 'customized' data type – we will learn a lot about these
  - Objects represent encapsulated, somewhat autonomous pieces of code with specified functionality
- Objects have:
  - **Properties** or **attributes**
  - **Functions** or **methods**



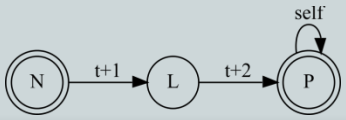
The Text object

- You can test an object being of a certain Class:

```
>>> isinstance(text1, Text)
True
```

- You can access properties of an object using . (dot) notation:

```
>>> text1.name
'Moby Dick by Herman Melville 1851'
```



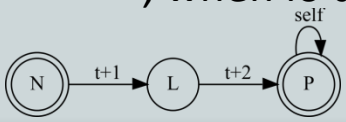
The Text object

- Objects can respond to many built-in functions:

```
>>> len(text1)
260819
```

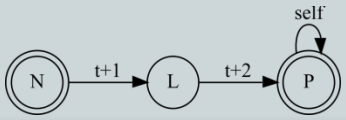
- But they also have their own special functions, **methods**, accessed with ***.name(arguments)***
- The method ***.concordance()*** takes a String argument:

```
>>> text1.concordance("harpoon")
Displaying 25 of 76 matches:
hen they were nigh enough to risk a harpoon from the bowsprit ? Now having a ni
n a sunrise and a sunset . And that harpoon -- so like a corkscrew now -- was f
 over the fire - place , and a tall harpoon standing at the head of the bed . B
, when lo and behold , he takes the harpoon from the bed corner , slips out the
```



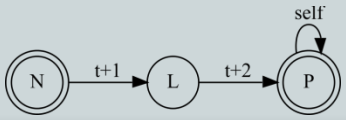
Methods and arguments

- Method arguments can be optional, in which case a default is supplied
- .concordance() can take 3 arguments:
  - word (a Unicode string, the word being searched for)
  - width (characters to display, default = 79)
  - lines (how many results maximum, default = 25)



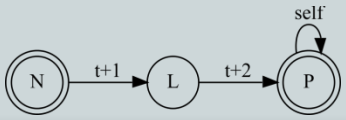
Methods and arguments

```
>>> text1.concordance("harpoon",10,10)
Displaying 10 of 76 matches:
risk a harpoon
And that harpoon
a tall harpoon
…

>>> text1.concordance("harpoon",lines=100)
Displaying 76 of 76 matches:
hen they were nigh enough to risk a harpoon from the bowsprit ? Now having a ni
n a sunrise and a sunset . And that harpoon -- so like a corkscrew now -- was f
 over the fire - place , and a tall harpoon standing at the head of the bed . B
```



First notions about OOP

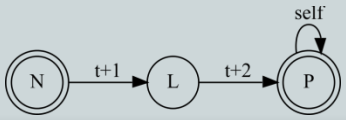
- A major line of thought for Object Oriented Programming (OOP)
  - Objects are encapsulated
  - They do their job and expose methods
  - We don't know (and don't want to know) ***how***
- Advantages:
  - Others can import our objects without studying our code
  - Possible to improve our objects without altering their **interface** to the outside



An example: *.similar(word)*

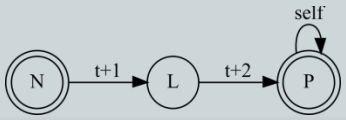
- The Class Text has a *.similar* function with the **signature**: .similar(word, num=20)
- It gives you ***num*** distributionally similar words
  - How?
  - Who cares: The **Text** Class takes care of this for us
  - If we have a better method to do this next week, we'll release a new version of the **Text** Class

- This is nice and correct in principle… but stay vigilant!



An example: *.similar(word)*

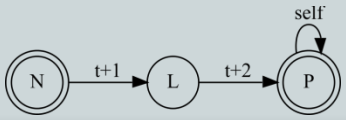
```
>>> text1.similar("boat",num=4)
whale ship head sea
>>> text1.similar("Ahab",num=4)
it he that queequeg
>>> text1.similar("crew",num=4)
whale ship head boat
>>> text1.similar("harpoon",num=4)
whale boat ship sea
```



A (slightly) more serious program

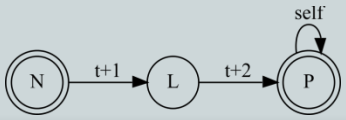
- For our next exercise, we will build a program to check whether our input is a **palindrome:**
  - dud
  - kayak

- Or not:
  - bud
  - magic



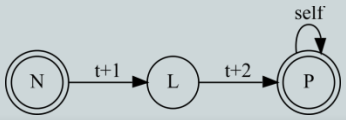
Palindrome checker

- Thinking about input and output:
  - IN: a string of characters
  - OUT:
    - An answer in English (String)
    - Or maybe: True or False (Boolean)



Some starter code

```
test = "kayak"

# Ideally we'd want something like this:
if test_is_a_palindrome:
        print("The input '" + test + "' is a palindrome")
else:
        print("The input '" + test + "' is not a palindrome")


# We need to learn more…
```

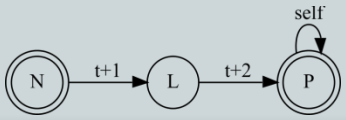


A word about indentation

- To know what to do '*only **if*** X' Python uses indentation:

```
x = 5                    # Not indented, always do this part
y = user_input           # Also not indented
if x > y:  # Not indented, since this check always happens
   print("it's bigger!") # This is indented – only do if x>y
```

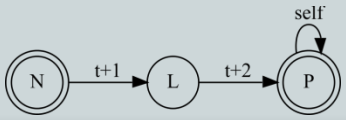
- In other words, indentation determines the **scope** of the **if** statement



If, else and elif

- You can also test multiple alternatives:

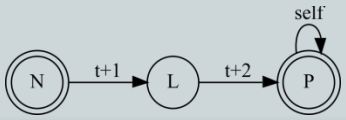
```
x = 5
y = user_input
if x > y:
   print("it's bigger!")
elif x < y:
   print("it's smaller!")
else:
   print("it's the same!")
```



Quick exercise – imagine it's snowing!

- Hurray! Snow!!
- What will this code print?

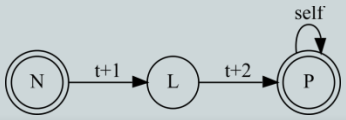
```
snow_inches = 40                # Current snow level
campus_open_max = 55            # Level at which campus closes
tomorrow_min = 10               # Minimum projected snowfall tomorrow
tomorrow_max = 20               # Maximum project snowfall tomorrow

if snow_inches + tomorrow_max < campus_open_max:
    print("campus will definitely be open")
elif snow_inches + tomorrow_min < campus_open_max:
    print("campus might be open")
else:
    print("campus will definitely be closed")
```



Another word about indentation

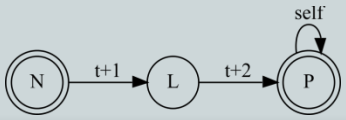
- Python accepts two ways of indenting:
  - Initial spaces, often 4 (sometimes 2 are used)
  - Tabs
- **PEP8** recommends 4 spaces
- But many developers use tabs (more in EU)
- I will accept either, but no mixing!

- What is PEP?



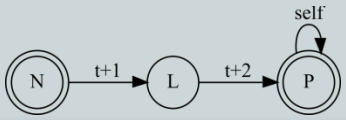
Conventions and names

- It is a good idea to use informative names for variables and functions
- To document your code inside your scripts
  - Helps others work with your code
  - Helps you to remember what you were doing
  - Allows creation of automatic documentation
- High quality code is easy to maintain – but what conventions should we use?



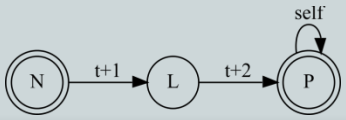
PEP

- Python is developed using the Python Enhancement Proposal (PEP) process
  - Enhancements to the language in newer versions (e.g. adding new operators, built in functions…)
  - Various **recommendations**
- Crucial for learning to write readable code:
  - PEP 0008: Style Guide for Python Code https://www.python.org/dev/peps/pep-0008/
  - PyCharm automatically checks PEP8 compliance
  - We will follow PEP8 in our assignments



Some PEP8 basics

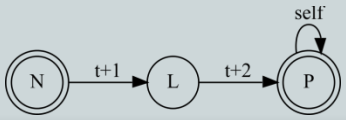
- Variables and functions should have informative, lower case names:
  - ✓ word_count ✗ wdct
  - ✓ find_nouns() ✗ findnn(), FindNouns()
- White space around operators:
  - ✓ count = previous + 1
  - ✗ count=previous+1
- Line length should be 79 characters maximum



Indentation and hierarchy

- Note that indentation is hierarchical:

```
if x > y:                # Not indented, always happens.
→if x > y * 2:           # Indented, happens if x > y
→→print("it's a lot bigger!")     # Indented twice!
→else:
→→print("it's a bit bigger")     # Indented twice!
else:
→print("it's smaller")
```



For next time: NLTK practice

- Please work through the NLTK book (Python 3.X version), chapter 1, through the end of section 1: (up to **"A Closer Look at Python"**)
  - http://www.nltk.org/book/ch01.html
- We'll review some of the functions in class later
  - Note: for the dispersion_plot, you'll need to install NumPy and Matplotlib using ***pip install numpy*** etc.
  - No need to submit anything – but ask a TA or me for help if you get stuck!