# Python Programming for Linguists

## Week 4

Shu-Kai Hsieh

LOPE lab at Graduate Institute of Linguistics,
National Taiwan University

1. Agenda

2. Python Programming: Functions and Modules
   - Review

3. Linguistic Analysis via NLTK
   - Simple Lexical Statistics
   - Customize your Corpus Raw Data

4. Homework

- **[Lecture]**: Functions (and Modules)
- **[Praxis]**: In-class exercises with NLTK
- **[Homework]**: see below
- **[Reading Assignment]**: Python book: chapter 3-4 (Reading and Scripting)

# Functions and Modules

[Ref] Learning with Python, 2nd Edition (Chapter 3).

- Introduction: Defining a Function
- Function Parameters and Arguments
- Local Variables and Global Statement
- Default Argument Values
- Keyword Arguments
- The RETURN statement:Using the literal statement

# Saving and Executing Your Programs

- The interactive interpreter is one of Python's great strengths. It makes it possible to test solutions and to experiment with the language in real time. If you want to know how something works, just try it!
- However, everything you write in the interactive interpreter is lost when you quit. What you really want to do is write programs that both you and other people can run.

# Saving and Executing Your Programs

Steps (Skip this if you are using `Canopy`)

- First of all, you need a text editor, preferably one intended for programming.
- If you are already using IDLE, simply create a new editor window with `File` → `New Window`. Another window appears — without an interactive prompt.
- Start by entering `print 'Hello, world!'`, now select `File` → `Save` to save your program (e.g. `hello.py`) (Be sure to put it somewhere where you can find it later on, or you might want to create a directory where you put all your Python projects, such as `C:python` in Windows.)
- Now you can run it with `Edit` → `Run` script, or by pressing `Ctrl-F5`.

# Saving and Executing Your Programs

Now let's practice it by extending our script to the following, save and run.

## Example

```
name = raw_input('What is your name?')
print 'Bonjour, ' + name + '!'
```

# Running Your Python Scripts from a Command Prompt

Actually, there are several ways to run your programs.

**assume that** you have a DOS window or a UNIX shell prompt before you, and that the Python executable (called python.exe in Windows, and python in UNIX) has been put in your PATH environment variable. Also, let's assume that your script from the previous section (hello.py) is in the current directory.

Then you can execute your script with the following command:

## Example

```
C:\>python hello.py    # in Windows

$ python hello.py      # in UNIX
```

# Making Your Scripts Behave Like Normal Programs

In Unix/Linux/Mac

**Example**

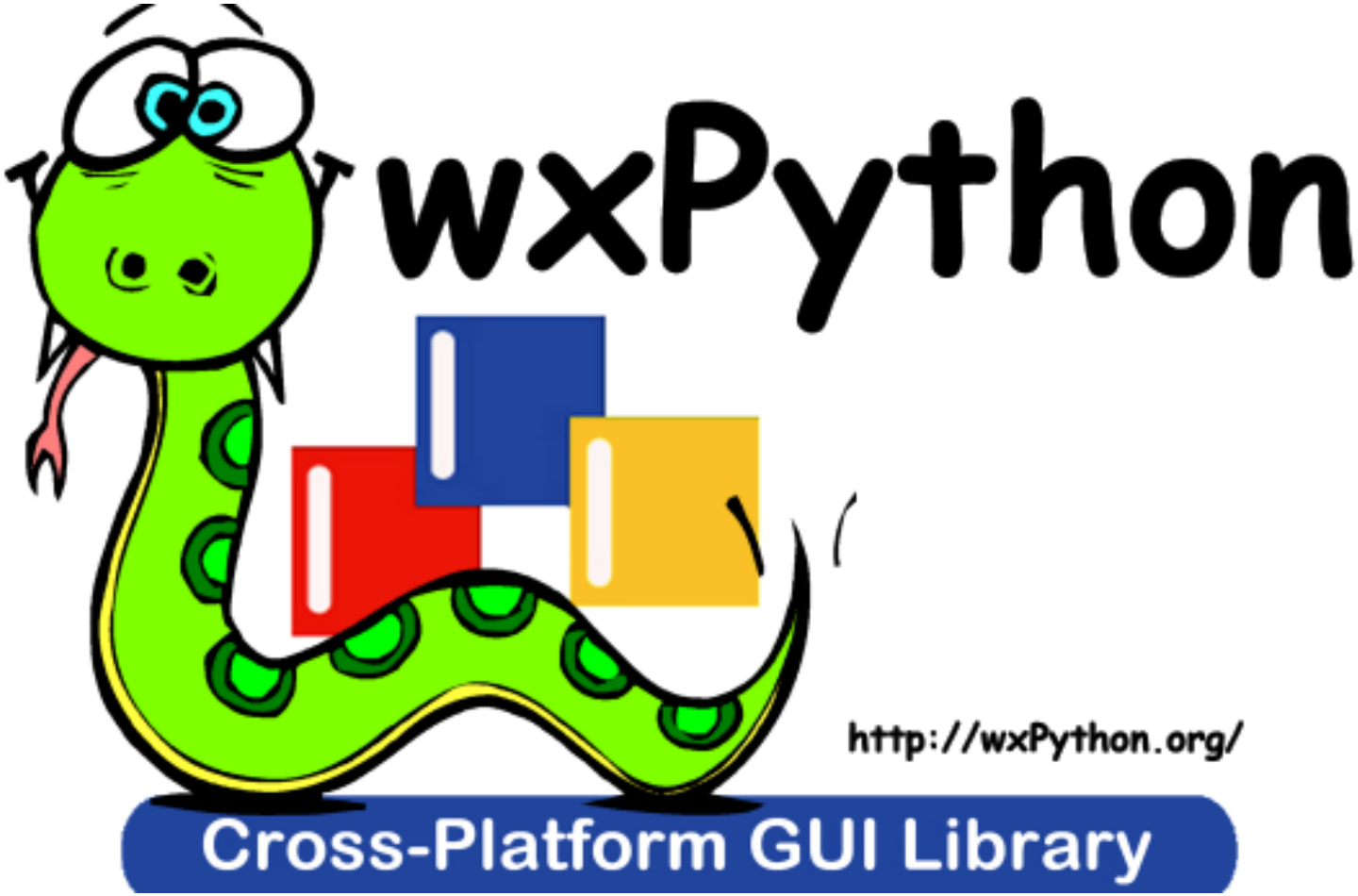
```
#!/usr/bin/env python   # put this in the first line of your scri

chmod a+x hello.py     # Before you can actually run your script,
hello.py  # now run the script
```

What about double clicking in Windows?

Try double-clicking the file hello.py, what happens?

# Making Your Scripts Behave Like Normal Programs

Or, make your programs look better, with buttons, menus, and so on!



# Introduction

- A Function is a block of *reusable* codes that you can use to perform a specific action.
- Calling the function: you can give a name to the block of codes and you can run that block using that name anywhere in your program and any number of times.
- Defining a function: Functions are created in Python using the keyword **def**, and the name of the function followed by the code that will executed when the function is used.
- The code is separated from the function's name by a colon and should be indented (i.e., **4 spaces**). The names of functions are subject to the same constraints as the names of variables.

## Example

```
#!/usr/bin/python


def speakOut():
  ''' my first proposal script'''
   print "Will you marry me?"  # block belonging to the function
                               # End of function
speakOut()                     # call the function
```

# Function Parameters and Arguments

- We define a function called `speakOut()`. This function takes no parameters.
- Parameter: A name used inside a function to refer to the value passed as an argument.
- Argument: A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.
- and hence in `speakOut()` there are no variables declared in the parentheses.

# Function Parameters and Arguments

- For now, think of parameter(s) as variables and argument(s) as values. These parameters are just like variables except that the values of these variables are defined when we call the function and are not assigned values within the function itself. Parameters are specified within the pair of parentheses in the function definition, separated by commas.
- When we call the function, we supply the values in the same way.

# Now let's get back to `speakOut`

## Example

```
def speakOut(candidate):
    '''My second proposal script'''
    print "Will you marry me," candidate, "?"



speakOut("Owen")
speakOut("Shukai")
speakOut("Taco")
speakOut()
help(speakOut)
```

# Variables and Parameters are Local!

When you create a variable inside a function definition, it is not related in any way to other variables with the same names used outside the function, i.e., it is **local** to the function, existing only inside the function. This is called the *scope* of the variable.

# Now let's get back to `speakOut`

## Example

```
def speakOut_twice(candidate1, candidate2):
    '''My second proposal script'''
   print 'Will you marry me?', candidate1,'and',candidate2,'?'
    candidate3 = candidate1 + candidate2
    print 'Or, will you marry me?', candidate3, '?'

speakOut_twice('Taco', 'Owen')

# what about this?
print candidate3

# When speakOut_twice terminates, the variable cat is destroyed.
```

# Local Variables

- In the function, the first time that we use the value of the name x, Python uses the value of the parameter declared in the function. Next, we assign the value 2 to x. The name x is local to our function. So, when we change the value of x in the function, the x defined in the main block remains unaffected. In the last print statement, we confirm that the value of x in the main block is actually unaffected.
- **Using the global statement**: If you want to assign a value to a name defined outside the function, then you have to tell Python that the name is not local, but it is global. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

## Example

```
def func():
      global x

      print 'x is', x
      x = 2
      print 'Changed global x to', x

x = 50
func()
print 'Value of x is', x
```

# Local Variables

- The global statement is used to declare that x is a global variable, hence, when we assign a value to x inside the function, that change is reflected when we use the value of x in the main block. You can specify more than one global variable using the same global statement. For example, global x, y, z.

# Default Argument Values

- For some functions, you may want to make some of its parameters as optional and use default values if the user does not want to provide values for such parameters. This is done with the help of default argument values.
- You can specify default argument values for parameters by following the parameter name in the function definition with the assignment operator (=) followed by the default value. Note that the default argument value should be a constant.

## Example

```
def shout(message, times = 1):
    print message * times
shout('Hello')
shout('World', 5)
```

# Keyword Arguments

- If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called keyword arguments - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two advantages - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

## Example

```
def keyarg(a, b=6, c=100):
print 'a is', a, 'and b is', b, 'and c is', c


keyarg(2, 88)
keyarg(233, c=124)
keyarg(c=509, a=300)
```

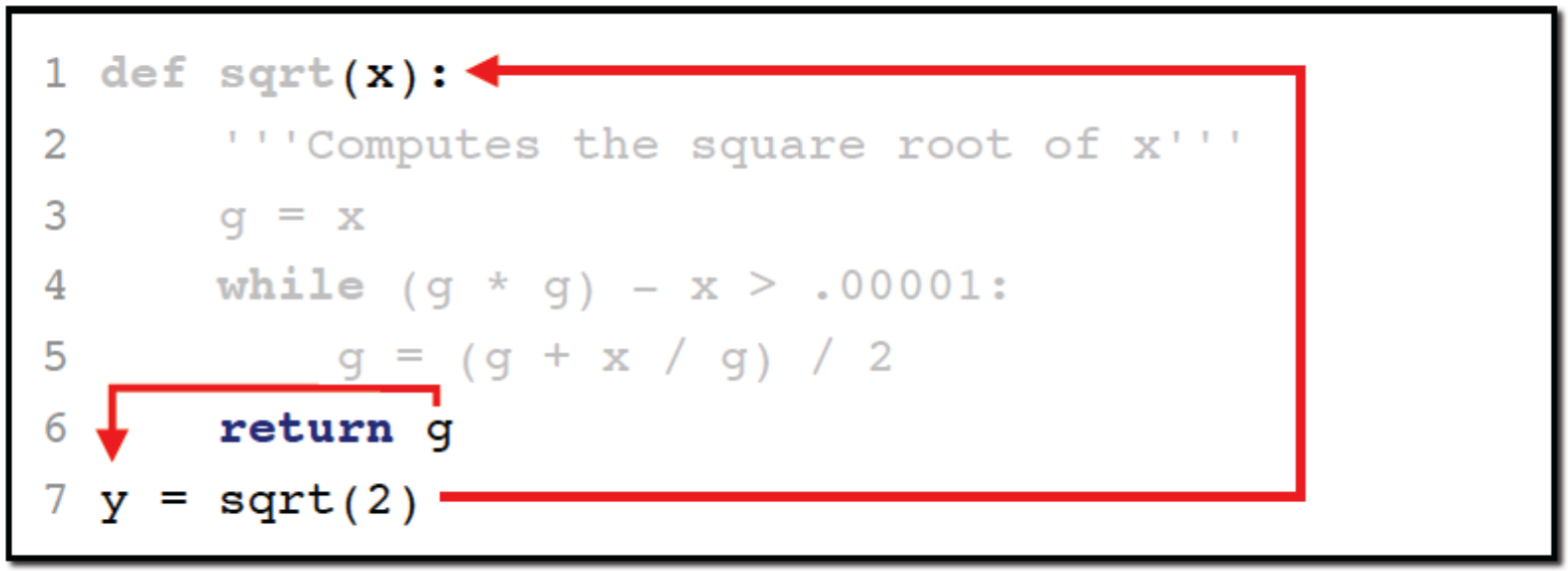
# The `return` Statement

The return statement is used to return from a function i.e. break out of the function. We can optionally return a value from the function as well.

## Example

```
def maximum(x, y):
    if x > y:
        return x
    else:
        return y

print maximum(2, 3)
```

# Function Application (for Semanticist)




- When the function is called, the parameters are instantiated with the values from the function call.
- The function call evaluates to the value returned by the function.

# Modules

- A module is a *file* that contains a collection of related *functions*.
- Modules serve as extensions that can be imported into Python to extend its capabilities.
- You import modules with a special command called (naturally enough) `import`.
- A collection of related modules is called a package.
- A set of packages is sometimes called a library (e.g., NLTK is a library.)

# Modules

dot notation: The syntax for calling a function in another module by specifying the module name followed by a dot(period) and the function name.

## Example

```
import math


degrees = 45
radians = degrees/360.0 * 2 * math.pi
math.sin(radians)
```

1. Agenda

2. Python Programming: Functions and Modules
   - Review

3. Linguistic Analysis via NLTK
   - Simple Lexical Statistics
   - Customize your Corpus Raw Data

4. Homework

# Simple Lexical Statistics

Lexical statistics with NLTK

1. Exploration of the ways we can bring our computational resources to bear on large quantities of text.
2. What makes a text distinct, and use automatic methods to find characteristic words and expressions of a text.
    - Frequency Distributions
    - Fine-Grained Selection of Words
    - Collocations and Bigrams
    - Counting Other Things

# Frequency Distributions

**Frequency Distributions** : tells us how the total number of word tokens in the text are distributed across the vocabulary items. More specifically, we want to find

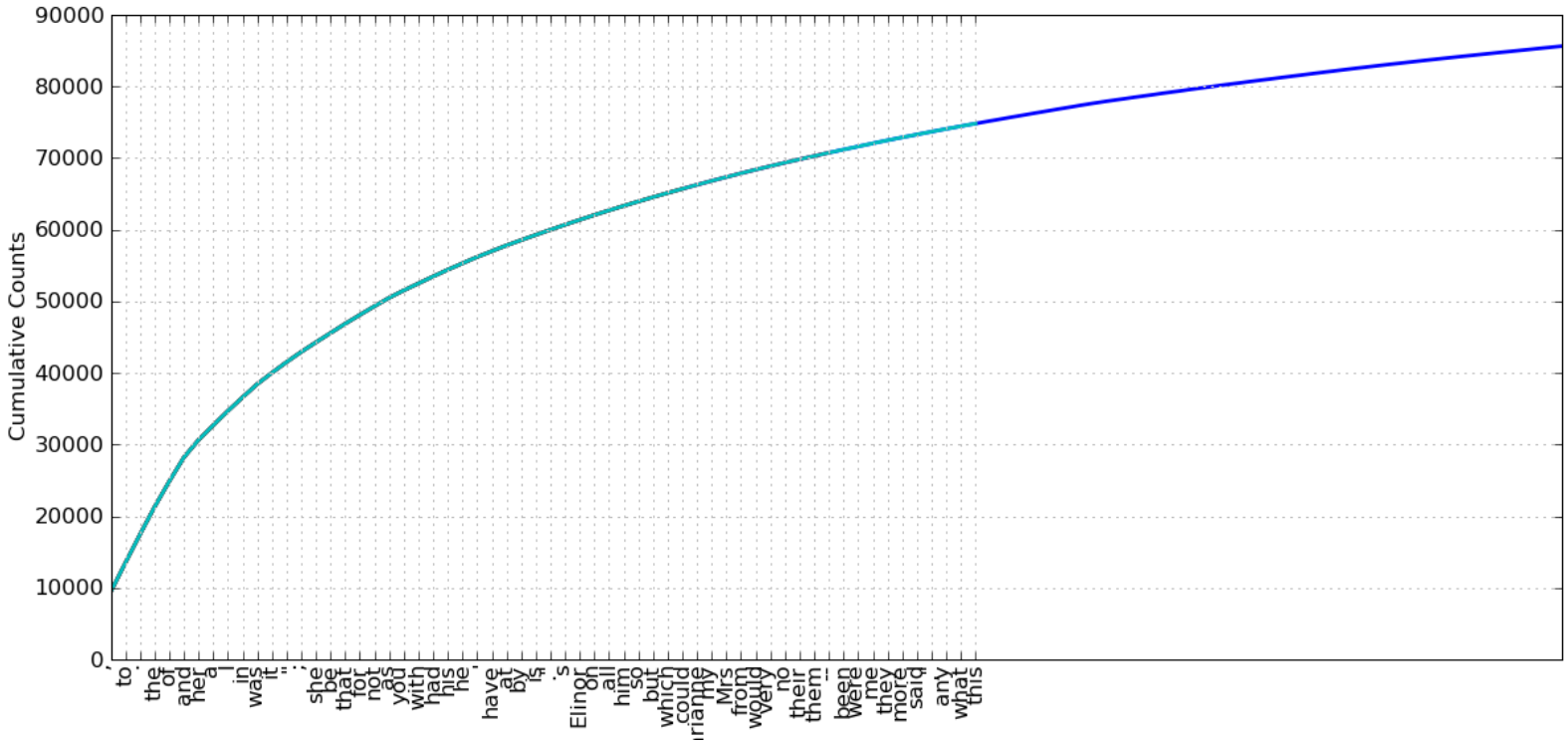
- the 60 most frequent words of '*Sense and Sensibility*'(Jane Austen, 1811)
- the total word numbers that have been counted up.
- a list of distinct word types in the text.
- the frequency of a certain word in the text
- cumulative frequency (plot)
- the hapaxes (the words that occur once pnly)

## Example

```
from nltk.book import *  # DO NOT forget this


fdist1 = FreqDist(text2)
fdist1
vocabulary1 = fdist1.keys()
vocabulary1[:60]
fdist1['what']
fdist1.plot(60, cumulative=True)
fdist1.hapaxes()  # how many?
```

# Cumulative frequency plot of Sense and Sensibility

What can you say from the figure?



# Functions defined for frequency distributions in NLTK

*Table 1-2. Functions defined for NLTK's frequency distributions*

| Example | Description |
|---|---|
| `fdist = FreqDist(samples)` | Create a frequency distribution containing the given samples |
| `fdist.inc(sample)` | Increment the count for this sample |
| `fdist['monstrous']` | Count of the number of times a given sample occurred |
| `fdist.freq('monstrous')` | Frequency of a given sample |
| `fdist.N()` | Total number of samples |
| `fdist.keys()` | The samples sorted in order of decreasing frequency |
| `for sample in fdist:` | Iterate over the samples, in order of decreasing frequency |
| `fdist.max()` | Sample with the greatest count |
| `fdist.tabulate()` | Tabulate the frequency distribution |
| `fdist.plot()` | Graphical plot of the frequency distribution |
| `fdist.plot(cumulative=True)` | Cumulative plot of the frequency distribution |
| `fdist1 < fdist2` | Test if samples in `fdist1` occur less frequently than in `fdist2` |

# Fine-Grained Selection of Words

For each word `w` in the Vocabulary `V`, we check whether `len(w)` is greater than 10.

## Example

```
V = set(text2)
long_words = [w for w in V if len(w) > 10]
sorted(long_words)

# what if I wnat to get frequently occurring long words?
sorted([w for w in set(text2) if len(w) > 7 and fdist1[w] > 7])
```

# Collocations and Bigrams

- Collocations: a sequence of words that occur together unusually often.
- To get a handle on collocations, we need to extract from the texts a list of word pairs (**bigrams**), or word triplets (**trigrams**), etc. Check with Google Book N-gram Viewer !!

# Collocations and Bigrams

## Example

```
text2.collocations()
bigrams(['more', 'is', 'said', 'than', 'done'])
```

# Counting Other Things

- The distribution of word lengths in a text
- Creating `FreqDist` out of a long list of numbers, where each number is the length of the corresponding word in the text.

# Counting Other Things

## Example

```
# First derive a list of the lengths of words in text2
[len(w) for w in text2]


# counts the num. of times each of these occur
fdist = FreqDist([len(w) for w in text2])


fdist.keys()
fdist.items()
fdist.max()
fdist[4]
fdist.freq(4)
```

# Exercise

## Writing a Function

- Write a function with the name `lexical_statistics(my_text)` that takes a parameter `my_text` and returns the word tokens, vocabulary size and lexical diversity score. The text should be loaded with the help of `open()`, and needs to be *tokenized* with `nltk.word_tokenize()` .
- To read a local file, you will need to know a built-in `open()` function, followed by the `read()` method. Suppose that you got a file *alice.txt*, you can load its contents like this:

# How to tokenize a text?

## Example

```
f=open('alice.txt')
raw=f.read()


tokens = nltk.word_tokenize(raw) # here!
type(tokens)
tokens[:11]
```

# Got that, now what I want is this:

```
f = open('alice.txt')
# f = open(r'C:\text\alice.txt')
raw = f.read()
raw_new = nltk.word_tokenize(raw) # here!

lexical_statistics(raw_new)



word tokens: 31555
vocabulary size: 3894
diversity score: 8.10349255265
```

# Exercise: Solution

## Example

```
from __future__ import division


def lexical_statistics(my_text):
    my_text_data = nltk.word_tokenize(my_text)
    word_tokens = len(my_text_data)
    vocab_size = len(set(my_text_data))
    diversity_score = word_tokens / vocab_size
    print 'word tokens:', word_tokens
    print 'vocabulary size:', vocab_size
    print 'diversity score:', diversity_score
```

# Exercise: Solution (even BETTER!)

Even more powerful when combining with linux and R scripting !(More later)

**Example**

```
cat alice.txt | python lexical_statistics.py | sort |.....
```

# Loading your own texts

To use the methods of NLTK for your texts, load them with the help of NLTK's `PlaintextCorpusReader`.

## Example

```
from nltk.book import *
from nltk.corpus import PlaintextCorpusReader
corpus_root      =      '/ Users/ shukai/ Uni_works/ NTU/
pythonProgramming2011-12/data_tools_apis/data'
myTexts = PlaintextCorpusReader(corpus_root, ['alice.txt',
            'dostoevsky.txt'])
myTexts.fileids()
```

## Example

```
myTexts.words()    # myText.words('alice.txt')
myTexts.words()[0:12]
myTexts.sents()
```

# Make your text an NLTK text!

## Example

```
myText4NLTK = nltk.Text(tokens)
myText4NLTK[0:111]
myText4NLTK.collocations()

myText4NLTK.concordance("rather")
myText4NLTK.concordance("very")
myText4NLTK.common_contexts(['rather','very'])
```

## Example

```
len(myText4NLTK)
len(set(myText4NLTK))


from __future__ import division
len(myText4NLTK)/len(set(myText4NLTK))
myText4NLTK.count('dance')
```

## Example

```
fdist = FreqDist(myText4NLTK)
vocabulary = fdist.keys()
vocabulary[:40]
fdist['very']
fdist.plot(50,cumulative=True)
```

# Advanced Scripting in Bigrams

## Example

```
nltk.bigrams(myText4NLTK)
myBigrams= nltk.bigrams(myText4NLTK)
bi_fdist = FreqDist(myBigrams)
bi_vocabulary = fdist.keys()
bi_vocabulary[:40]


bi_fdist[('went', 'on')]
bi_fdist.plot(50,cumulative=True)
```

# Homework

1. Read Learning with Python: [Chapter 4 Conditionals] and NLTK-book: [Chapter 2]
2. (60%)

Define a function called `myTextFunc(text)` that has a single parameter for the text, and which returns the token numbers, type numbers and lexical diversity rate of the 3 texts.

| texts | tokens types lexical diversity |
|---|---|
| alice.txt | |
| dostoevsky.txt | |
| hamlet.txt | |

3. (40%) Write a function to compare the usage of modal verbs between ANC/BNC.