# Python Programming for Linguists

## Week 5

Shu-Kai Hsieh

LOPE lab at Graduate Institute of Linguistics,
National Taiwan University

1. Agenda
2. Reviews on Modules
3. Conditionals and Recursion
   - Boolean Values and Expressions
4. Linguistic Analysis via NLTK
   - Review: Customize your Corpus Raw Data

- **[Lecture]**: Conditionals and Recursion
- **[Praxis]**: In-class exercises with NLTK
- **[Homework]**: see below
- **[Reading Assignment]**: Python book: chapter 5-6 (Reading and Scripting)

# Reviews on Modules

- A module is a *file* that contains a collection of related *functions*.
- You import modules with a special command called (naturally enough) `import`.
- 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. (These functions are usually known as methods).
- A collection of related modules is called a package; set of packages is sometimes called a library (e.g., NLTK is a library.)

# Reviews on Modules

- When looking for a module, Python will look for a file named module.py in the directories listed on `PYTHONPATH`, which can be viewed the built-in path:

```
import sys
sys.path
```

- It is easy to make a file that can be used as a module (and thus imported), or as a script. You wrap the statements in a function, typically called main, and then have an if statement at the end that says if run as a script, run main.

# Reviews on Modules

## Example

```
#!/usr/bin/python
""" This file is both a script and a module """
def hello_world():
    print "Hello!world"
if __name__ == "__main__": hello_world()
```

# Reviews on Modules

## Example

```
# Save this as 'mymodule.py' in the path.
# Now we can use it either as a script,


$ ./mymodule.py
Hello!world


# or imported as a module:
>>> import mymodule
>>> mymodule.hello_world()
# and alternatively,
>>> from mymodule import hello_world
>>> hello_world()
```

# Conditionals and Recursion

[Ref] HTTLCS (Chapter 4).

- Modulus operator
- Boolean expressions and logical operators
- Conditional execution
- Alternative execution
- Chained and nested conditionals
- Recursion
- Keyboard input

# Control-Flow in Python

Python provides a complete set of control-flow elements, with conditionals and loops.

- The `if-elif-else` statement
- The `while` and `for` loop

# General Form of Conditionals

```
if condition1:
    body1
elif condition2:
    body2
elif condition3:
    body3
.
.
.
elif condition(n-1):
    body(n-1)
else:
    body(n)
```

- The body after the `if statement` is required. But you can use the **pass statement** here (as you can anywhere in Python where a statement is required). The `pass statement` serves as a placeholder where a statement is needed, but it performs no action:

```
if x < 5:
    pass
else:
    x = 5
```

- There is no `case` (or `switch`) statement in Python.

## Example

```
luckyNumber = 66
guess = int(raw_input('Please enter an integer :'))


if guess == luckyNumber:
    print 'Congratulations, you guessed it.'
elif guess < luckyNumber:
    print 'No, it is a little higher than that'
else:
    print 'No, it is a little lower than that'
print 'Done'
```

# Writing a guessing function that earns bucks!

Let's first have a quick look at the built-in `random()` module. To import it, we use:

```
>>> import random
```

We can use the **choice** function in the random module to select a random element from a **list**:

**Example**

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> random.choice(a)
6
```

Another efficient way to do this is to use a function called 'randint' that takes two integer arguments (`inclusive, exclusive`) and returns a random integer between those two integers. For example,

**Example**

```
>>> random.randint(1, 11)
```

will return a random integer between 1 and 10.

# Exercise

Now Let's use this function to create a guessing game function. The function will first select a random number and then allow the user to guess the number. If the user's guess is incorrect it will keeping asking and giving hints to the user until the correct number is guessed.

```
def guess():
""" Guessing function """

    [[Try it now on your own]]
```

# More challenging exercise: What's Your Lucky Word Today

- You've written a function with the name like `myLexFun(my_text)` that takes a parameter `my_text` and returns the word tokens, vocabulary size and lexical diversity score.
- Now revise your function (and make it a module), so that you can get your lucky word from alice.

```
from myLexFun() import luckyword
luckyword()
```

# The while Loop

```
while condition:
    body
else:
    post-code
```

As long as the expression is `True`, the body will be executed repeatedly. If it evaluates to False, the while loop will execute the post-code section and then terminate. (the else part of the while loop is optional and not often used, though)

```
num = 10
while num > 0:
    print(num)
    num = num - 1     # num -= 1
```

# The for Loop

- In Python, a `for` loop iterates over the values returned by any iterable object—that is, any object that can yield a sequence of values.
- Sometimes you need to loop with explicit indices (to use the position at which values occur in a list). You can use the `range` command together with the `len` command on lists to generate a sequence of indices for use by the for loop.

```
# This code prints out all the positions in a list
# where it finds negative numbers:
x = [1, 3, -7, 4, 9, -5, 4]
for i in range(len(x)):
    if x[i] < 0:
        print("Found a negative number at index ", i)
```

# List and dictionary comprehensions

The pattern of using a for loop to iterate through a list, modify or select individual elements, and create a new list or dictionary is very common.

```
x = [1, 2, 3, 4]
x_squared = []
for item in x:
    x_squared.append(item * item)
```

Alternative code that does exactly the same thing:

```
x = [1, 2, 3, 4]
x_squared = [item * item for item in x]
```

# List Comprehension

- This sort of situation is so common that Python has a special shortcut for such operations, called a (list) comprehension, with the pattern:

```
new_list = [expression for variable in old_list if expression]
```

- You can think of it as an example of Python **idiom**, a fixed notation that we use habitually without bothering to analyze each time.

# List Comprehension: More examples (NLTK 1.4)

**Example**

```
from nltk.book import *
sorted([w for w in set(text1) if w.endswith('ableness')])
sorted([term for term in set(text4) if 'gnt' in term])
sorted([item for item in set(text6) if item.istitle()])
```

1. Agenda
2. Reviews on Modules
3. Conditionals and Recursion
   - Boolean Values and Expressions
4. Linguistic Analysis via NLTK
   - Review: Customize your Corpus Raw Data

- Python has a Boolean object type that can be set to either True or False. Any expression with a Boolean operation will return `True` or `False`.
- 0 or empty values are `False`, and any other values are `True`.

# Comparison and Boolean Operators

Normal operators `<`, `<=`, `>`, `>=`, `==`, `!=`

Membership testing operators : `in`, `not in`; `is`, `is not`.

Boolean operators `and`, `or`, `not`

# Comparison and Boolean Operators: Example

Suppose we want to find *frequently occurring long words* in *Sense and Sensibility by Jane Austen*(`text2`).

**Example**

```
fdist2 = FreqDist(text2)
sorted([w for w in set(text2) if len(w) > 7 and fdist2[w] >7])
```

# Challenging Exercise: Create a wc-like script and Make it a Module

Write a script that roughly replicates the UNIX `wc` utility and reports the number of **lines, words**, and **characters** in a file.

```
#!/usr/bin/env python
""" Reads a file and returns the number of lines, words,
and characters - similar to the UNIX wc utility
"""
# Opens file
infile = open('alice.txt')
#  Reads  file;  splits  into  lines  (Ref:  NLTK-
book Table 3-2.)
lines = infile.read().split("\n")
% ######## YOUR CODE HERE###############
```

1. Agenda

2. Reviews on Modules

3. Conditionals and Recursion
   - Boolean Values and Expressions

4. Linguistic Analysis via NLTK
   - Review: Customize your Corpus Raw Data

1. Agenda
2. Reviews on Modules
3. Conditionals and Recursion
   - Boolean Values and Expressions
4. Linguistic Analysis via NLTK
   - Review: Customize your Corpus Raw Data

# 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

```
% tokens = nltk.word_tokenize(myTexts)

myText4NLTK = nltk.Text(myTexts)
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)
```