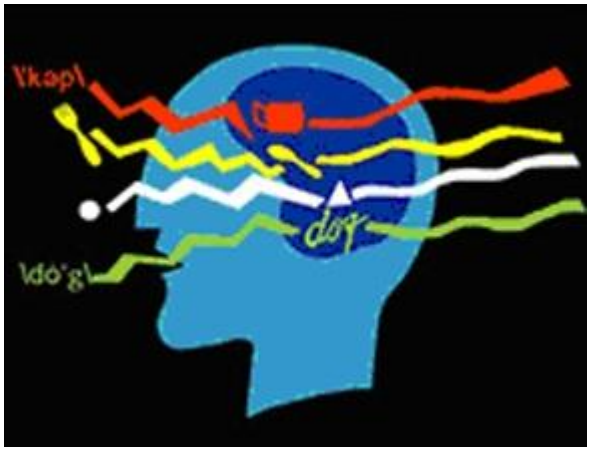
Topic 2:

Python for Natural Language Processing (Pt1)



What is Python?

- Python is an open source scripting language
- Developed by Guido van Rossum in the early 1990s
- Named after *Monty Python (a comedy series)*
- Available for download from http://www.python.org

Why is Python Suitable for NLP?

- **Very Object Oriented**
  - Python much less verbose than Java
- **NLP Processing: Symbolic**
  - Python has built-in data types for strings, lists, and more.
- **NLP Processing: Statistical**
  - Has strong numeric processing capabilities: matrix operations, etc.
  - Suitable for probability and machine learning code.
- **NLTK: Natural Language Tool Kit**
  - Widely used for teaching NLP
  - Implemented as a set of Python modules
  - Provides adequate libraries for *many* NLP building blocks

Why is Python Suitable for NLP?

- Interpreted language
  - works with an evaluator for language expressions
- Dynamically typed
  - variables do not have a predefined type
- Rich, built-in collection types
  - Lists
  - Tuples
  - Dictionaries (maps)
  - Sets
- Concise

Features

- Indentation instead of braces
- Newline separates statements
- Several sequence types
  - Strings **'...'**: made of characters,
  - Lists **[...]** : made of anything,
  - Tuples **(...)** : made of anything,
- Powerful subscripting (*slicing*)
- Functions are independent entities (not all functions are methods)
- Exceptions as in Java

The Python Interpreter/Shell

- Interactive interface
  - Command line or IDE

```
Python 3.5.2 (default, Apr 11 2012, 07:12:16) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

  - Evaluating inputs

```
Python 3.5.2 (default, Apr 11 2012, 07:12:16) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 3*(7*2)
42
>>>
```

Sequence types: Tuples, Lists, and Strings

- Tuple
  - A simple immutable ordered sequence of items
  - Items can be of mixed types, including collection types
- Strings
  - Immutable
  - Conceptually very much like a tuple
  - (8-bit characters. *Unicode strings* use 2-byte characters)
- List
  - Mutable ordered sequence of items of mixed types

Sequence types: Tuples, Lists, and Strings

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.

- Key difference:
  - Tuples and strings are **immutable**
  - Lists are **mutable**

Sequence Types 1

- Tuples are defined using parentheses () and commas.

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes (", ', or """").

```
>>> st = "Hello World"
>>> st = 'Hello World'
```

- String that uses triple quotes."""

```
>>> st = """This is a multi-line"""
```

Sequence Types 2

- We can access individual members of a tuple, list, or string using square bracket “array” notation. Index starts from 0.

```
          0     1     2       3       4
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

Positive and negative indices

```
          0     1       2       3       4
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

```
         -5     -4     -3      -2      -1
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Negative lookup: count from right, starting with –1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members.

- Start copying at the first index, and stop copying *before* the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
  ['abc', 45, 4.34, 23]
```

- We can change lists *in place.*
- Name ***li*** still points to the same memory reference when we're done.

Tuples: Immutable

```
>>> tup = (23, 'abc', 4.56, (2,3), 'def')
>>> tup[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item
 assignment
```

You can't change a tuple.

You can make a fresh new tuple and assign its reference to a previously used name.

```
>>> newtup = tup
```

- The immutability of tuples means they're faster than lists.

Operations on Lists(1)

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a')
# Note the method syntax

>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

Operations on Lists(2)

```
>>> li = ['a', 'b', 'c', 'b']

>>> li.index('b')
# index of first occurrence* 1
```

* More complex forms exist

```
>>> li.count('b')
# number of occurrences 2

>>> li.remove('b')
# remove first occurrence
>>> li
['a', 'c', 'b']
```

extend vs *append*

- + creates a fresh list (with a new memory reference)
- extend takes a list as an argument.
- append takes a singleton (a single value) as an argument

```
>>>li
[1, 11, 'i', 3, 4, 5, 'a']

>>> li.extend([9, 8, 7])
>>> li
[1, 11, 'i', 3, 4, 5, 'a', 9, 8, 7]

>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists (3)

```
>>> li = [5, 2, 6, 8]

>>> li.reverse() # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()       # sort the list *in place*
>>> li
  [2, 5, 6, 8]
```

Summary: Tuples vs. Lists

- Lists are slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- To convert between tuples and lists use the list() and tuple() functions:

```
>>> li = list(tu)
>>> tu = tuple(li)
```

Dictionaries: A *Mapping* type

- Dictionaries store a *mapping* between a set of keys and a set of values.
  - Keys can be any *immutable* type and unique.
  - Values can be any type and non-unique
  - Values and keys can be of different types in a single dictionary
  - Can also be used to define a set
- You can
  - define
  - modify
  - view
  - lookup
  - delete

Creating and accessing dictionaries

```
>>> d = {} #create an empty dictionary
>>> d = {'user':'joe', 'pswd':1234}

>>> d['user']
'joe'    # displays the 'value' of the key 'user'

>>> d['pswd']
1234

>>> d['joe']
# no such key : 'joe' is a value and not key

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: joe
```

Creating and accessing dictionaries

```
>>> ages = { "Sam " :4, "Mary " :3, "Bill " :2 }
>>> ages
{'Sam ': 4, 'Bill ': 2, 'Mary ': 3}
>>> for name in ages.keys():
        print(name, ages[name])

Sam 4
Bill 2
Mary 3
```

Updating Dictionaries

- Dictionaries are unordered
  - New entry might appear anywhere in the output.
  - Dictionaries work by *hashing*

- Keys must be unique.
- Assigning to an existing key replaces its value.

```
>>> d = {'user':'joe', 'pswd':1234}
>>> d['user'] = 'jay'
>>> d
{'user':'jay', 'pswd':1234}

# adds new key and value to dictionary
>>> d['id'] = 45
>>> d
{'user':'jay', 'id':45, 'pswd':1234}
```

Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> del d['user']          # Remove one.
>>> d
{'p':1234, 'i':34}

>>> d.clear()              # Remove all.
>>> d
{}
>>> a =[1,2]
>>> del a[1]# (del also works on lists)
>>> a
[1]
```

Opening & closing a file

- Reading from a file:
  - open( ) : returns a file object, and commonly takes two arguments: open(filename, mode).

```
>>> f = open('sentence.dat', 'r')
>>> print(f)
<open file 'sentence.dat', mode 'r' at
80a0960>
```

  - close( ) : close connection of file

```
>>> f.close()
```

Reading from a file (cont…)

- Reading from a file:
  - read() : reads the entire content of a file or some quantity of data
  - readline( ) : returns a single line from a file
  - arguments: empty or size in bytes

```
>>> f.read()
This is the first sentence of the
paragraph.\n''This is the second sentence
of the paragraph.\n‘

>>> f.readline()
This is the first line of the file.\n'
```

Reading from files (cont…)

- Reading from a file using a loop:
  - line by line

```
for line in f:
    print(line)

This is the first line of the file.
This is the second line of the file.

with open('pride_and_prejudice.txt') as f:
    s = f.read()
    print(s)
```

Reading from a text file

- Reading from a file using a loop:
  - line by line

```
for line in open("recipe_ital_102.txt"):
    for word in line.split():
      if word.endswith("ing"):
         print(word)

frying
baking
```

Writing into files (cont…)

- Writing into a file:
  - write( ) : write argument into file,
  - arguments: string
  - may need to convert values into string first

```
>>> f = open(‘prob.dat', ‘w')
>>> print(f)
<open file ‘prob.dat', mode ‘w' at 80a0960>
>>> f.write('This is a test\n')
>>> value = ('the answer is', 42)
>>> s = str(value)
>>> f.write(s)
```

Appending items into files

- Appending into a file:
  - if we use the mode ‘w’, existing contents will be erased

```
>>> f = open(‘prob.dat', ‘a')
>>> f.write('This is a test\n')
>>> value = ('the answer is', 42)
>>> s = str(value)
>>> f.write(s)

This is a test
The answer is 42
```

Text as List of Words

- What is a text?
  - Sequence of symbols
  - Sequence of **words and punctuations**
  - Sequence of paragraphs, sequence of sections chapters….
- Words in a text is usually stored as a list in Python
- A text can be separated into words (tokenized) using split()
- Tokenized words (in a list) can be concatenated into a string using .join()

```
>>> text = "Natural Language Processing with Python"
>>> text.split()
['Natural', 'Language', 'Processing', 'with', 'Python']
>>> ' '.join(text.split())
'Natural Language Processing with Python'
```

Importing Modules (os)

- *os* module: for keeping track of current directory and files
  - os stands for *operating system*

```
>>> import os
>>> os.chdir('c:/users/student/')
>>> infile = open('paragraph.dat','r')
>>> inread = infile.readlines()
```

Functions are first-class objects in Python

- Functions can be used as any other data type
- They can be
  - Arguments to function
  - Return values of functions
  - Assigned to variables
  - Parts of tuples, lists, etc
  - …

Calling a Function

- The syntax for a function call is:

```
>>> def myfun1(x, y):
            return x * y
>>> myfun1(3, 4)
   12

>>> def myfun2(x):
        return x*3
>>> myfun2(2)
   6

>>> def applier(q, x):
        return q(x)
>>> applier(myfun2, 7)
21
```

Writing a Complete Function to Manipulate a Text File (Example)

```
#recipe.txt
import os
os.chdir('e:/sem2-1617/CSC 4309/data/') #working dir

def find_word(filename):
    for line in open(filename): #open file
        #split sentences into words
        for word in line.split():
            #find words ending with 'ing'
            if word.endswith("ing"):
                print(word)
```

Exercise 1

- *Use the input files (Italian recipes) uploaded in Google Classroom and perform the following:*

1. *Write a function that read the contents of recipe_ital_115.txt and does the following :*
   - *Create a dictionary that defines each unique word the files and stores its frequency*
   - *Count how many times the word "of" occurs in the file*
   - *Find words that end with 'ly' in the file*
   - *Print all your output in a file named 'results.txt'*