Inf2a: Lab 2

Object-Oriented Programming and NLTK

1 Object-Oriented Programming

A class is defined using the keyword `class`, followed by the class name. The class body must be indented, and the first statement can be a documentation string.

A method is a function that "belongs to" a class. Methods must have, as their first parameter, the variable `self`, that contains the reference to the object itself (for example `def greet(self, name)`: It is equivalent to the variable `this` in Java, with the difference that you need to define it explicitly.

By default, all methods and attributes in Python are public. It is possible to make them 'pseudo' private, adding two underlines before their name (for example as in `__func(self)`).

⋆ Type the lines introduced by `>>>` and by `...`

Remember that indentation matters in Python, which means that in the following example, you cannot simply hit `<Enter>` on the third line; you will need to type the appropriate number of spaces/tab first.

```
>>> class MyClass:
...     """A very simple class"""
...
...     def greet(self, name):
...         return "Hello, " + name
...
```

A class is instantiated into an object, and methods of the object are called like in Java.

⋆ Type the lines introduced by `>>>` and by `...`

```
>>> c = MyClass()
>>> c.greet("Paolo")
```

Classes can have an initialisation method `__init__()` similar to the class constructor in Java. This method is called when the class is instantiated and can have a set of parameters. In contrast with Java, that can have many different

constructors, there can be a only one such methods per class.

⋆ Type:

```
>>> class Greeter:
...     """A simple class"""
...     def __init__ (self, greeting):
...         self.greeting = greeting
...     def greet(self, name):
...         return self.greeting + ", " + name
...
>>> c2 = Greeter("hi")
>>> c2.greet("tim")
```

A class can derive from another class:

⋆ Type:

```
>>> class GreeterEx(Greeter):
...     """A derived class"""
...     def bye(self):
...         return "Bye Bye"
...
>>> c3 = GreeterEx("hello")
>>> c3.greet("mr smith")
>>> c3.bye()
```

This class will contain the methods defined in `Greeter`, plus the new `bye()` method.

2 Passing Parameters

It is possible to pass parameters to a script.

⋆ Create in your editor a file named `test.py`

⋆ Type in the editor:

```
import sys
for arg in sys.argv:
    print arg
```

⋆ Save the file

⋆ Type in the shell:

```
python test.py these are the arguments
```

The arguments are stored in the variable `sys.argv`, that is a list of string. `sys.argv[0]` contains the name of the script, while the following elements contains the arguments.

3 Natural Language Toolkit (NLTK)

NLTK is a suite of Python libraries and programs for symbolic and statistical natural language processing. It has extensive documentation, including tutorials that explain the concepts underlying the language processing tasks supported by the toolkit (`http://nltk.org/`).

⋆ Open a terminal window, and go to the folder "`MyPython`" that you created during Lab 1.

⋆ Launch the python shell using the command `python`.

To load the NLTK libraries we use the `import` statement.

⋆ Type:

```
>>> from nltk.tokenize import simple
```

A description of a class is available using the `help` function.

⋆ Type:

```
>>> help(simple)
```

Type `q` to finish the help mode. You can check the documentation of the NLTK API to see all available modules and classes in:

`http://nltk.org/api/nltk.html`

4 Tokens

For most kinds of linguistic processing, we need to identify and categorise the words of the text. This turns out to be a non-trivial task. Here, we introduce *tokens* as the building blocks of text, and show how texts can be tokenized.

⋆ Type:

```
>>> from nltk.tokenize import WhitespaceTokenizer
>>> text = 'Hello world! This is a test string.'
>>> WhitespaceTokenizer().tokenize(text)
```

As you can see, tokenization based on whitespace alone is not sufficient. The method `tokenize.regexp` uses a regular expression to determine how text should be split up. This regular expression specifies the characters that can be included in a valid word.

⋆ Type:

```
>>> from nltk.tokenize import RegexpTokenizer
>>> text = "Hello. Isn\'t this fun?"
>>> pattern = r'\w+|[^\w\s]+'
>>> tokenizer = RegexpTokenizer(pattern)
>>> tokenizer.tokenize(text)
```

The statement:

`|r'\w+|[^\w\s]+'`

creates a regular expression which accepts sequences formed by "word" characters, i.e., characters other than whitespace or punctuation (the subexpression `\w+`) or by characters which are neither word characters nor whitespace characters (`[^\w\s]+`, the sign `^` means complement). The previous regular expression is not good enough with the string $22.40 and 44.50%, where we might want to keep the symbol $ and % attached to the number.

⋆ Type:

```
>>> text = 'That poster costs $22.40.'
>>> tokenizer.tokenize(text)
```

The union of the previous regular expression with this one: `\$\d+\.\d`, where `\d` represents a decimal digit, solves the first problem.

⋆ Type:

```
>>> pattern = r'\w+|\$\d+\.\d+|[^\w\s]+'
>>> tokenizer = RegexpTokenizer(pattern)
>>> tokenizer.tokenize(text)
```

A note on the union operator (|):

A|B, where A and B are both regular expressions, creates a regular expression that will match either A or B. An arbitrary number of regular expressions can be separated by the '|' in this way. As the target string is scanned, regular expressions separated by '|' are tried from left to right. When one pattern completely matches, that branch is accepted. This means that once A matches, B will not be tested further, even if it would produce a longer overall match. In other words, the '|' operator is never greedy. To match a literal '|', escape it using \, or enclose it inside a character class, as in [|].

For more information about how to create regular expressions check:

`http://http://docs.python.org/2/library/re.html#re-syntax.`

5 Corpora

NLTK is distributed with several corpora. They can be accessed using the `corpus` package.

First we import the Brown Corpus, the first million word, part-of-speech tagged electronic corpus of English. Each of the sections *a* through *r* represents a different genre. List of available sections can be accessed using the `items` and `documents` variables.

⋆ Type:

```
>>> from nltk.corpus import brown
>>> brown.categories()

>>> brown.fileids()

>>> help(brown)
```

With the method `sents` we can access tokenized sentences of a corpus text and count the number of them.

⋆ Type:

```
>>> count=0
>>> for sntc in brown.sents('cn01'):
...     print sntc
...     count += 1
...
>>> print count
```

The first argument of the `sents` method is the fileid of the corpus. Try the example with `'ca02'`.

In order to extract a specific sentence, we can use the index.

⋆ Type:

```
>>> print brown.sents('cn01')[0]
```

With the `tagged_sents` method we can access the tagged text of a corpus. Every word in the sentence is tagged and the sentence is presented as a list of binary tuples containing the word and its tag.

⋆ Type:

```
>>> print brown.tagged_sents('cn01')[0]
```

Since the structures accessed through `corpus` are basic Python structures, we can use them to do some analysis on a corpus. For instance, we can count the number of times each part-of-speech tag occurs in the Brown corpus.

⋆ Type:

```
>>> dict = {}
>>> for fileid in brown.fileids():
...     for sntc in brown.tagged_sents(fileid):
...         for word,tag in sntc:
...             if tag in dict:
...                 dict[tag]+= 1
...             else:
...                 dict[tag]=0
...
```

```
>>> for tag,count in dict.items():
...     print tag," ",count
...
```

We can use a `ConditionalFreqDist` to find the most frequent occurrence of a word in a context, for instance the previous word. First, we pass each token of the corpus together with its previous token to the `ConditionalFreqDist` class. We call this process *training*.

⋆ Type:

```
>>> from nltk.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()

>>> prev=None
>>> for fileid in brown.fileids():
...     for sntc in brown.sents(fileid):
...         for token in sntc:
...             cfdist[prev][token] += 1
...             prev=token
...
```

Notice that the first token of the corpus has no "previous word". For this example, use `None` as the context for the first token.

We can see the words which follow another word using the `samples()` method.

⋆ Type:

```
>>> cfdist['living']
```

Try this with a different word of your own choice.

We can use the information stored in `ConditionalFreqDist` to create a text generator using the most frequent word given a word. The following code creates a sentence of word length 20 starting with 'an'. The most frequent word is given by `max()`.

Please note that this will generate strings which may not be well-formed sentences.

⋆ Type:

```
>>> word = 'an'
>>> for i in range(20):
...     print word,
...     word = cfdist[word].max()
...
```

Try this with a different word.

6 Grammars

The `grammar` module defines a set of classes that can be used to define context free grammars:

- The `grammar.Nonterminal` class is used to represent nonterminals.
- The `grammar.Production` class is used to represent (CFG) productions.
- The `grammar.ContextFreeGrammar` class is used to represent CFGs.

`Nonterminal` is a simple class that is used to let NLTK distinguish terminals from nonterminals.

⋆ Type:

```
>>> from nltk import grammar as cfg
>>> S = cfg.Nonterminal('S')
>>> S

>>> NP = cfg.Nonterminal('NP')
>>> NP

>>> VP, Adj, V, N = cfg.nonterminals('VP, Adj, V, N')
>>> VP, Adj, V, N
```

Each `Production` specifies that a nonterminal (the left-hand side of a rule) can be expanded to the sequence of terminals and nonterminals given in the right-hand side of the rule.

⋆ Type:

```
>>> prod1 = cfg.Production(S, [NP, VP])
>>> prod1

>>> prod2 = cfg.Production(NP, ['the', Adj, N])
>>> prod2
```

Context free grammars are encoded by the `CFG` class. A CFG consists of a special start nonterminal, and an ordered list of productions.

⋆ Type:

```
>>> grammar = cfg.CFG(S,[prod1,prod2])
>>> grammar

>>> grammar.start()

>>> grammar.productions()
```

Additionally, with the `fromstring` function of the `CFG` class, it is possible to create a grammar from its text description.

⋆ Type:

```
>>> from nltk import CFG
>>> grammar2 = CFG.fromstring('''
...     S -> NP VP
...     NP -> "I" | "John" | "Mary" | "Bob" | Det N
...     VP -> V NP | V NP PP
...     V -> "saw" | "ate"
...     Det -> "a" | "an" | "the" | "my"
...     N -> "dog" | "telescope" | "apple"
...     PP -> P NP
...     P -> "on" | "with"
...     ''')

>>> grammar2

>>> grammar2.start()

>>> grammar2.productions()
```

For a grammar, we can parse a sentence and get its syntactic tree using the `RecursiveDescentParser()` function.

⋆ Type:

```
>>> from nltk import parse
>>> from nltk.tokenize import WhitespaceTokenizer
>>> from nltk.parse import RecursiveDescentParser
>>> sntc1 = WhitespaceTokenizer().tokenize('I saw Mary')
>>> sntc2 = WhitespaceTokenizer().tokenize('John ate my apple')
>>> rd_parser = RecursiveDescentParser(grammar2)

>>> for p in rd_parser.parse(sntc1):
...     print p

>>> for p in rd_parser.parse(sntc2):
...     print p
```

7 Treebank

NLTK also includes a 10% fragment of the Wall Street Journal section of the Penn Treebank. Each sentence of the corpus is available in three forms; (1) as tokenized text, (2) as tokens labelled with part of speech, and (3) as parse trees. These can be accessed using `treebank.raw()` for the raw text, `treebank.tagged()` for the tagged text, and `treebank.parsed()` for the parse trees.

⋆ Type:

```
>>> from nltk.corpus import treebank
>>> help(treebank)
```

```
>>> print treebank.raw('wsj_0001.mrg')

>>> print treebank.words('wsj_0001.mrg')

>>> print treebank.tagged_words('wsj_0001.mrg')

>>> print treebank.parsed_sents('wsj_0001.mrg')[0]
```

The argument in the above three functions, `'wsj_0001'`, is a section of the treebank. We can see the list of all sections using `treebank.items`. The function `treebank.parsed()[]` returns an object of class `nltk.Tree`. We can analyse the tree using different functions available in this class.

⋆ Type:

```
>>> t = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> t.label()

>>> t.leaves()

>>> print t

>>> len(t)

>>> t[0]

>>> t[0].label()

>>> len(t[0])
```

`t.label()` contains the top-most node in the tree `t`. We can see the list of words that yields the parse tree using `t.leaves()`. Using `len(t)`, we can get the number of child nodes to `t.label()`. Each of these child nodes and their subtrees can be accessed using `t[0].label()` and `t[0]` respectively.

Using the above utilities, we can find out the subject of any given sentence. The following code prints the subject NP phrase in the first sentence of `'wsj_0001'`

⋆ Type:

```
>>> t = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> for ch_tree in t:
...     if (ch_tree.label().startswith('NP-SBJ')):
...         print ch_tree.leaves()
```

8 Exercises

EXERCISE 1

Create a class named `Queue` that models a queue: the first element that enters is the first that exits (FIFO: First In, First Out). The class will use a list to maintain the data. It will expose the following methods:

- `isempty()`: verifies if the queue is empty
- `push(item)` inserts an element at the end of the queue
- `pop()`: extracts and returns the first element in the queue (possibly only if the queue is not empty)

Import the module into the python shell, and test it Remember to create the list that contains the data before accessing to it.

EXERCISE 2

Using your preferred editor, create a class named `Stack` that models a stack: the last element that enters is the first that exits (LIFO: Last in, First Out). The class willl use a list to maintain the data. It will expose the following methods:

- `isempty()`: verifies if the stack is empty
- `push(item)`: inserts an element at the end of the stack
- `pop()`: extracts and returns the last element of the stack (possibly only if the stack is not empty)

Import the module into the python shell, and test it. Remember to create the list that contains the data before accessing to it.

EXERCISE 3

Using your favourite editor, create a file that defines a class `Checker` that contains the function you wrote in Exercise 5 of Lab 1. Save the file as "`oo_checker.py`" in your "`MyPython`" directory. Then import your new module from the python shell, instantiate the class into an object and call the method.

EXERCISE 4

Using your preferred editor, create a class that checks if a string (`infix`) is contained in other strings. The class must be initialised passing the `infix` string, that must be stored in a class variable. The class must expose the method `check(string)` that verifies if `infix` is contained in the passed string (you can use the operator `in` to verify if a string is contained in another one:
`string1 in string2`).

Hint Remember to use `self` when trying to access to class methods and attributes.

Import your module into the python shell, and test its behaviour (you must instantiate the class passing the infix string, and then call the method check passing different strings)

Hint If you use the statement `import modulename`, remember to use the modulename prefix in front of the class name. If you make an error in the class, and you need to reimport the module, use `reload(modulename)`. `import` will not reimport a module already imported. You will also have to reinstantiate the class.

EXERCISE 5

Create a class for managing a phone book. The user must be able to:

- insert a name and the relevant phone number,
- obtain a number from a name,
- verify if name is in the book,
- list all the names and phone numbers in the book,
- delete a name from the book
- as optional feature, the user should be able to print in alphabetical order the names and their phone numbers

Import your class into the python shell, and test it (remember to instantiate the class).

Hint Use a dictionary to store the data, and remember to create the it before using it. You can use the method `keys()` to obtain the list of all the keys. Then you can apply any method available for the lists on the list you have obtained.

EXERCISE 6

Add to the pattern:

```
r'\w+|\$\d+\.\d+|[^\w\s]+'
```

the option to tokenize percentages as a single tokens, for instance:

- 100.00%, 10.5%, and 10.234%.

Hint Use the union operator (|) to add your option to the start of `pattern`. Test with the following text:

```
>>> text = 'The interest does not exceed 8.25%.'
```

EXERCISE 7

Use the following code as a starting point:

```
>>> from nltk.corpus import brown
>>> from nltk.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()

>>> prev=None
>>> for fileid in brown.fileids():
...     for sntc in brown.sents(fileid):
...         for token in sntc:
...             cfdist[prev][token] += 1
...             prev=token
...

>>> cfdist['living']

>>> word = 'an'
>>> for i in range(20):
...     print word,
...     word = cfdist[word].max()
...
```

With 'an', the text generator gets stuck in a cycle within the first 20 words. Modify the program to choose the next word randomly from a list of the most likely words in the given context. (Hint: store the most likely words in a list `lwords`, and then randomly choose a word from the list using `random.choice(lwords)`. Don't forget to `import` the `random` module. In order to find the most likely words, use `sorted()` instead of `samples()`).

- Remember that we saw at the start of this section that each part of the Brown Corpus corresponds to a different genre of English. For example, Part j corresponds to scientific text. Select a genre of your choice from the Brown Corpus. Train your system on this corpus and get it to generate random text. You may have to experiment with different start words.
- Try the same approach with a different genre.
- Compare the generated texts. How do the resulting texts differ?.

EXERCISE 8

Create a new grammar (`grammar3`) using as a base grammar2 (from section 6) plus the production `NP -> Det N PP`.

grammar2:

```
S -> NP VP
NP -> "I" | "John" | "Mary" | "Bob" | Det N
VP -> V NP | V NP PP
V -> "saw" | "ate"
```

```
Det -> "a" | "an" | "the" | "my"
N -> "dog" | "telescope" | "apple"
PP -> P NP
P -> "on" | "with"
```

- Parse the sentence *Mary saw the dog with the telescope*. How many parse trees do you get?
- For `grammar2`, write down two unambiguous sentences (that yields just one tree), two ambiguous sentences (that yields more than one tree), and two ungrammatical sentences (that yields no tree at all).

EXERCISE 9

```
>>> t = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> for ch_tree in t:
...     if (ch_tree.label().startswith('NP-SBJ')):
...         print ch_tree.leaves()
```

- Extend the above program to identify the subject in all the sentences in `'wsj_0003'`.
- A subordinate clause in a sentence will have its own subject. So, extend the code (using recursion) to identify all the subjects in every sentence.