# PyNLPI: Python Natural Language Processing Library

Maarten van Gompel

2-12-2011

# First things first

**PyNLPL** is pronounced as...

# Introduction

## What is PyNLPI?

**Python Natural Language Processing Library**

- A collection of custom-made Python modules usable in Natural Language Processing
- Very modular setup
- Reusable object-oriented modules, prevents "reinventing the wheel" for common tasks
- Using PyNLPI enables you to more quickly write NLP tools, as you need not start from scratch.

# Installation

## Installation

PyNLPl is on github: `http://github.com/proycon/pynlpl`
To obtain it: `$ git clone`
`https://github.com/proycon/pynlpl`

# Questions and Answers (1/4)

**Q:** Why reinvent the wheel yourself and not use for example NLTK?
**A:** Firstly because there are many customised modules not present in NLTK, such as modules for dealing with FoLiA, D-Coi, Timbl, Cornetto, DutchSemCor. Secondly, because reimplementing things myself was a good learning process to better understand certain algorithms.

# Questions and Answers (2/4)

**Q:** How did PyNLPI came to be?
**A:** Often code is (and should be) modular and reusable in the future. Whenever that is the case, I put it into PyNLPI.

# Questions and Answers (3/4)

**Q:** Where is PyNLPl used?
**A:** In almost everything I write: PBMBMT, Valkuil, the DutchSemCor Supervised-WSD system heavily rely on PyNLPl.

# Questions and Answers (4/4)

**Q:** Why Python?
**A:** Elegant, modern and powerful scripting language, short development time. Great for text processing, easy to learn. Substantial user-base and 3rd party libraries available.

# Packages and modules in PyNLPI (1/3)

## Packages and modules in PyNLPI (1/3)

- `pynlpl.statistics` – Module containing classes and functions for statistics
- `pynlpl.evaluation` – Module for evaluation and experimentation, such as computation of precision/recall, creation of confusion matrices, etc.. Also contains abstract experiment classes and Wrapped Progressive Sampling.
- `pynlpl.datatypes` – Module containing data types
- `pynlpl.search` – Module containing search algorithms
- `pynlpl.textprocessors` – Module containing text processors

# Packages and modules in PyNLPI (2/3)

## Packages and modules in PyNLPI (2/3)

- `pynlpl.formats` – Contains modules for reading/writing specific file formats
  - `pynlpl.formats.timbl` – Module for reading Timbl output format
  - `pynlpl.formats.sonar` – Module for reading the SoNaR corpus (D-Coi XML)
  - `pynlpl.formats.folia` - Module for reading/manipulating/writing FoLiA XML
  - `pynlpl.formats.giza` - Module containing class for reading GIZA A3 alignment
  - `pynlpl.formats.moses` - Module containing class for reading phrase translation tabel
  - `pynlpl.formats.cgn` – `pynlpl.formats.dutchsemcor`

## Packages and modules in PyNLPl (3/3)

- `pynlpl.clients` – Contains network clients for various services.
  - `pynlpl.clients.cornetto` – Client to connect to Cornetto webservice
  - `pynlpl.clients.frogclient` – Client to connect to Frog server
- `pynlpl.lm` – Language Models
  - `pynlpl.lm.lm` – Contains simple language model
  - `pynlpl.lm.srilm` – SRILM module
  - `pynlpl.lm.server` – Generic LM Server

## Using pynlpl.statistics: FrequencyList and Distribution

```
>>> from pynlpl.statistics import FrequencyList, Distribution
>>> freqlist = FrequencyList()
>>> freqlist.append(['It','is','what','is','is'])
>>> freqlist
{'It': 1, 'it': 1, 'is': 2, 'what': 1}
>>> freqlist['is']
2
>>> dist = Distribution(freqlist)
>>> dist
{'It': 0.2, 'it': 0.2, 'is': 0.4, 'what': 0.2}
>>> dist['is']
0.4
>>> dist.entropy()
1.9219280948873623
```

## Creating N-grams with pynlpl.textprocessors.Windower

```
>>> from pynlpl.textprocessors import Windower
>>> s = ['It','is','what','it','is']
>>> list(Windower(s,2))
[('<begin>', 'It'), ('It', 'is'), ('is', 'what'),
('what', 'it'), ('it', 'is'), ('is', '<end>')]
```

## Combining things: Creating a tri-gram frequency list from a FoLiA document

```
>>> from pynlpl.formats import folia
>>> from pynlpl.statistics import FrequencyList
>>> doc = folia.Document(file='/path/to/folia_doc.xml')
>>> freqlist = FrequencyList()
>>> for trigram in Windower(doc.words(),3,False,False):
...         freqlist.count(trigram)
>>> freqlist.save('freqlist.txt')
```

## Creating a simple trigram Language Model of a corpus in FoLiA or DCOI XML

```
1 >>> from pynlpl.formats.folia import Corpus
2 >>> from pynlpl.statistics import FrequencyList
3 >>> simplelm = SimpleLanguageModel(3)
4 >>> for doc in Corpus('/path/to/for/example/sonar/')
5 ...     for sentence in doc.sentences():
6 ...         simpelm.append([ word.text() for word in
      sentence.words()])
7 >>> simplelm.save('sonar.trigram.lm')
```

## Using the Frog Client

First start Frog in server mode: `frog --skip=p -S 12345`

```
>>> from pynlpl.clients.frogclient import FrogClient
>>> client = FrogClient('localhost',12345)
>>> for word, lemma, morph, pos in client.process("Het is wat het is"):
...         print lemma, pos
het VNW(pers,pron,stan,red,3,ev,onz)
zijn WW(pv,tgw,ev)
wat VNW(vb,pron,stan,vol,3o,ev)
het VNW(pers,pron,stan,red,3,ev,onz)
zijn WW(pv,tgw,ev)
```

## Using the evaluation module (1/2)

```
>>> from pynlpl.evaluation import ClassEvaluation
>>> weather_forecast = ['sun','sun','rain','cloudy']
>>> actual_weather = ['cloudy','sun','rain','rain']
>>> evl = ClassEvaluation(actual_weather, weather_forecast)
>>> print evl
                TP FP TN FN Accuracy Precision Recall(TPR) Specificity(TNR) F-score
sun             1 1 2 0 0.750000 0.500000 1.000000 0.666667 0.666667
cloudy          0 1 2 1 0.500000 0.000000 0.000000 0.666667 0.000000
rain            1 0 2 1 0.750000 1.000000 0.500000 1.000000 0.666667

Accuracy                : 0.5
Recall      (macroav): 0.625
Precision   (macroav): 0.5
Specificity (macroav): 0.75
```

## Using the evaluation module (2/2)

```
>>> evl.confusionmatrix()
{('cloudy', 'sun'): 1, ('rain', 'cloudy'): 1, ('sun', 'sun'): 1, ('rain', 'rain'): 1}
>> print evl.confusionmatrix()
== Confusion Matrix == (hor: goals, vert: observations)

                cloudy  rain  sun
       cloudy        0     1    0
         rain        0     1    0
          sun        1     0    1
```

# Complex topics

## Search Algorithms

1. Define your search state, a class derived from the abstract class `pynlpl.search.AbstractSearchState`
2. Add methods `expand()` and for informed searches `score()`
3. Instantiate an initial search state
4. Pass this to the search algorithm of your choice, there are several implemented in `pynlpl.search`: DepthFirstSearch, BreadthFirstSearch, IterativeDeepening, BestFirstSearch, BeamSearch, HillClimbingSearch, StochasticBeamSearch
5. Obtain the solution(s) and/or path(s)

# Experiments

## Experiments

You can define your experiment, as a class derived from the abstract class `pynlpl.evaluation.AbstractExperiment`.
Overloading methods as `run()`, `start()`
You can then use these with `pynlpl.evaluation.ExperimentPool` for multi-threaded use and in `pynlpl.evaluation.ParamSearch` and `pynlpl.evaluation.WPSParamSearch` for parameter optimisation.

# Conclusion

## Contribute!

If you have a modular, re-usable, preferably object-oriented Python module useful for NLP tasks. Consider adding it to PyNLPI!

## Conclusion (shameless promotion)

- **Use** PyNLPI if it has modules you can use!
- **Contribute** to PyNLPI with new modules!