Python for NLP and the Natural Language Toolkit

Edward Loper

Python and Natural Language Processing

Python is a great language for NLP:

- Simple
- Easy to debug:
  - Exceptions
  - Interpreted Language
- Easy to structure:
  - Modules
  - Object Oriented Programming
- Powerful string manipulation

Modules and Packages

- Python *modules* "package program code and data for reuse." (Lutz)
  - Similar to *library* in C, *package* in Java.
- Python *packages* are hierarchical modules (i.e., modules that contain other modules).
- Three commands for accessing modules:
  1) `import`
  2) `from...import`
  3) `reload`

Modules and Packages: `import`

- The `import` command loads a module:

```
# Load the regular expression module
>>> import re
```

- To access the contents of a module, use *dotted names*:

```
# Use the search method from the re module
>>> re.search('\w+', str)
```

- To list the contents of a module, use `dir`:

```
>>> dir(re)
['DOTALL', 'I', 'IGNORECASE', ...]
```

Modules and Packages: `from...import`

- The `from...import` command loads individual functions and objects from a module:

```
# Load the search function from the re module
>>> from re import search
```

- Once an individual function or object is loaded with `from...import`, it can be used directly:

```
# Use the search method from the re module
>>> search('\w+', str)
```

Modules and Packages: `reload`

- If you edit a module, you must use the `reload` command before the changes become visible in Python:

```
>>> import mymodule
...
>>> reload(mymodule)
```

- The reload command only affects modules that have been loaded with `import`; it does not update individual functions and objects loaded with `from...import`.

`Import` vs. `from..import`

`Import`

- Keeps module functions separate from user functions.
- Requires the use of dotted names.
- Works with `reload`.

`from..import`

- Puts module functions and user functions together.
- More convenient names.
- Does not work with `reload`.

Regular Expressions

- Regular expressions are a powerful string manipulation tool.
- Use regular expressions to:
  - Search a string (`search` and `match`)
  - Replace parts of a string (`sub`)
  - Break strings into smaller pieces (`split`)

Regular Expression Syntax

- Most characters match themselves. For example, the regular expression "test" matches the string `'test'`, and only that string.
- [*x*] matches any *one* of a list of characters. For example, "[abc]" matches `'a'`, `'b'`, or `'c'`.
- [^*x*] matches any *one* character that is not included in *x*. For example, "[^abc]" matches any single character *except* `'a'`, `'b'`, or `'c'`.
- "." matches any single character.

Regular Expression Syntax (cont'd)

- *x** matches zero or more *x*'s. For example, "a*" matches `''`, `'a'`, `'aa'`, etc.
- *x*+ matches one or more *x*'s. For example, "a+" matches `'a'`, `'aa'`, `'aaa'`, etc.
- *x*? matches zero or one *x*'s. For example, "a?" matches `''` or `'a'`.
- *x{m,n}* matches *i x*'s, where $m \leq i \leq n$. For example, "a{2,3}" matches `'aa'` or `'aaa'`.

Regular Expression Syntax (cont'd)

- Parenthases can be used for grouping. For example, "(abc)+" matches `'abc'`, `'abcabc'`, `'abcabcabc'`, etc.
- *x*|*y* matches *x* **or** *y*. For example, "this|that" matches `'this'` and `'that'`, but not `'thisthat'`.

Regular Expression Syntax (cont'd)

- "\d" matches any digit; "\D" matches any non−digit.
- "\s" matches any whitespace character; "\S" matches any non−whtiespace character
- "\w" matches any alphanumeric character; "\W" matches any non−alphanumeric character
- "^" matches the beginning of the string; "$" matches the end of the string.
- "\b" matches a word boundry; "\B" matches position that is not a word boundry.

Introduction to NLTK

The Natural Language Toolkit (NLTK) provides:

- Basic classes for representing data relevant to natural language processing.
- Standard interfaces for performing tasks, such as tokenization, tagging, and parsing.
- Standard implementations of each task, which can be combined to solve complex problems.

NLTK: Top−Level Organization

- NLTK is organized as a flat hierarchy of packages and modules.
- Each module provides the tools necessary to address a specific task
- Modules contain two types of classes:
  - Data−oriented classes are used to represent information relevant to natural language processing.
  - Task−oriented classes encapsulate the resources and methods needed to perform a specific task.

NLTK: Example Modules

- **`nltk.token`**: processing individual elements of text, such as words or sentences.
- **`nltk.probability`**: modelling frequency distributions and probablistic systems.
- **`nltk.tagger`**: tagging tokens with supplemental information, such as parts of spech or wordnet sense tags.
- **`nltk.parser`**: high−level interface for parsing texts.
- **`nltk.chartparser`**: a chart−based implementation of the parser interface
- **`nltk.chunkparser`**: a regular−expression based surface parser

The Token Module

- It is often useful to think of a text in terms of smaller elements, such as words or sentences.
- The `nltk.token` module defines classes for representing and processing these smaller elements.

Tokens and Types

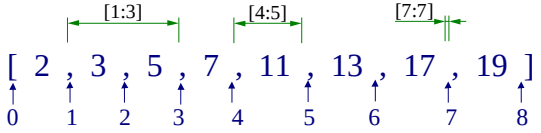
- The term *word* can be used in two different ways:
  1) To refer to an individual occurance of a word
  2) To refer to an abstract vocabulary item
- For example, the sentence "*my dog likes his dog*" contains five occurances of words, but four vocabulary items.
- To avoid confusion, use more precise terminology:
  1) ***Word token***: an occurance of a word
  2) ***Word type***: a vocabulary item

Text Locations

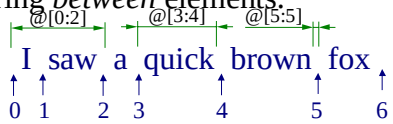
- A *text location* `@[s:e]` specifies a region of a text:
  - `s` is the *start index*
  - `e` is the *end index*
- The text location `@[s:e]` specifies the text beginning at `s`, and including everything up to (but not including) the text at `e`.
- This definition is consistant with Python *slice* notation.

Text Locations (continued)

- It is easiest to think of slice indices as appearing *between* elements.



- Similarly, you should think of location indices as appearing *between* elements:



Tokenization

- The simplest way to represent a text is with a single string.
- Difficult to process text in this format.
- Often, it is more convenient to work with a list of tokens.
- The task of converting a text from a single string to a list of tokens is known as *tokenization*.

Tokenization (continued)

- Tokenization is harder than it seems:

  I’ll see you in New York.

  The aluminum−export ban.

- The simplest approach is to use "graphic words" (i.e., separate words using whitespace)
- Another approach is to use regular expressions to specify which substrings are valid words.
- NLTK provides a generic tokenization interface: `TokenizerI`