Python Programming for Linguists

Week 10

Shu-Kai Hsieh

LOPE lab at Graduate Institute of Linguistics,
National Taiwan University

- **[Lecture]**: File I/O and Regular Expression
- **[Praxis]**: In-class exercises with NLTK
- **[Final Projects]**: see extra notes
- **[Reading Assignment]**: Read `Google's Python Class: Python Regular Expressions`

Standard I/O

So far you have seen two ways to display the contents of an object in python

- Using the `print` statement, which produces the most *human-readable* form of an object.
- Naming the variable at a prompt.

The `print` statement

```
x = 'python'
x
print x
```

Both of them are just strings, displayed for the benefit of the user. Try `x='你好'`

Built-in String Formatting

There are other ways (e.g., `pprint module`) to display an object as a string of objects, for the benefit of a human reader, or because we want to export our data to a particular format for other external program. E.g., given a frequency distribution `fdist`, we would do:

Example

```
fdist = nltk.FreqDist(['boy', 'boy', 'girl', 'boy', 'python'])
for word in fdist:
    print word, '->', fdist[word], ';',
```

Built-in String Formatting

- You see unwanted white spaces, and the print statement gets quite unreadable for more complex output.
- A better solution is to use **string formatting expressions**

Example

```
for word in fdist:
    print '%s->%d;' % (word, fdist[word]),
```

The % operator takes a string with **format specifiers** and an argument tuple. And %s, %d, %i are placeholders for strings, decimal strings, and integer numbers.

File I/O in Python

- Files are built-in objects
- Supports character- and line-based reading
- Encoded files with the `codecs` module

Use the `open()` function to open the file. You pass certain parameters to `open()` to tell it in which way the file should be opened - 'r' for read only, 'w' for writing only (if there is an old file, it will be written over). After use, files must be closed so that the content is actually written.

Example

```
infile = open('example.txt', 'r')
infile.read()
infile.close()
```

File objects support iteration

Print all lines in a file:

Example

```
for line in open('example.txt', 'r')
    print line
```

Encoded Files

What about Unicode scripts?

In Python 3, all strings are Unicode. In Python 2.x, you can use Unicode strings inside a .py script if you make Python aware of it. The first line of the script file should be a special comment:

Example

```
# -*- coding: utf-8 -*-
```

Encoded Files: Reading UTF-8 files

There is no magic function to detect the encoding of a file, so you need to know how it was stored in order to read it correctly. To read encoded files, you need to import `codecs` module, with its functions for opening

```
import codecs
f = open('/home/shukai/4Tutorial/plurk_sample.txt', 'r','UTF-8')

uni_string = f.read()
f.close()
print uni_string

# uni_string =open('/home/shukai/4Tutorial/plurk_sample.txt','r',encoding='utf-8').read()
```

Reading Chinese data (by converting it into Unicode)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


f = open('/home/shukai/4Tutorial/plurk_sample.txt')
s = f.read()
f.close()

# Use unicode() to brute-force converting a string to Unicode

uni_string = unicode(s,'utf-8')
for character in uni_string:
    print character

uni_string2 = uni_string.split()
for word in uni_string2:
    print word
```

Pickling objects into files!!

Python can write any data structure into a file and read that data structure back out of a file and re-create it, with just a few commands. This is an unusual ability but one that's highly useful. It can save you many pages of code that do nothing but dump the state of a program into a file (and can save a similar amount of code that does nothing but read that state back in).

$\rightarrow$ *Python provides this ability via the pickle module.*

Pickling objects into files!!

- Simple and powerful! Use **pickle** to store any Python object in a file and then get it back later intact. This is called storing the object persistently.
- There is another module called **cPickle** which functions exactly same as the pickle module except that it is written in the C language and is (up to 1000 times) faster. You can use either of these modules.

Pickling objects into files!!

Assuming that you have three variables in your program, and you can save them to a file (called `state`).

Example

```
import pickle


file = open("state", 'w')
# no matter what were stored in these variables,
# pickle.dump save everything!
pickle.dump(m, file)
pickle.dump(n, file)
pickle.dump(p, file)
file.close()
```

Restoring the pickled objects from files

To read that data back in on a later run of the program, just use `pickle.load()` to restore previous variables.

Example

```
import pickle


file = open("state", 'r')
m = pickle.load(file)
n = pickle.load(file)
p = pickle.load(file)
file.close()
```

Exercise [1] Pickling a list to a file

```
import cPickle as p
#import pickle as p

shoplistfile = 'shoplist.data'
# the name of the file where we will store the object
shoplist = ['apple', 'mango', 'carrot']

f = open(shoplistfile, 'w')
# Write to the file

p.dump(shoplist, f)
# dump the object to a file
f.close()

del shoplist
# remove the shoplist

f = file(shoplistfile)
# Write to the file

storedlist = p.load(f)
print storedlist
```

Pickle: Summary

- Use the `import .. as` syntax, which is handy since we can use a shorter name for a module. In this case, it even allows us to switch to a different module (cPickle or pickle) by simply changing one line!
- To store an object in a file, first we open a file object in write mode and store the object into the open file by calling the dump function of the pickle module. This process is called pickling.
- Next, we retrieve the object using the load function of the pickle module which returns the object. This process is called unpickling.

If your work involves storing or accessing pieces of data in large files, use shelve module that permits the reading or writing of pieces of data in large files, without reading or writing the entire file.

Introduction

- REs provide a mechanism for advanced text pattern matching, extraction, and/or search-and-replace functionalities.
- REs are simply strings containing *metacharacters* (special symbols and characters) that describe a pattern with which to recognize multiple strings (see the handout)

Introduction

Disjunction (OR): The concept of disjunction is expressed in RE with the vertical bar meta-character |

Example

```
a|an|the
air|spaceship
(air|space)ship
```

Character Classes

A string of characters (of any length) surrounded by square brackets which forms a pattern that matches any single character in that string.

Example

```
[a-zA-Z]
[a-zA-Z-]
[1-6]
```

Wildcard(DOT)

stands in for a *single* character. e.g., `p.n` matches *company, happened*, but not *planet, telephone*, etc.

Making literals out of Meta-characters: The Backslash

Example

```
etc.
etc\.
```

What happens when a literal character is escaped with a backslash? The answer is simple: Nothing!

Negation: The Caret in Square Brackets

By placing a caret at the beginning of a character range, you can ensure that a regular expression will NOT match any of the characters in it.

Example

```
for[^\.,!\?]
```

Special Sequences

Line Starts and Ends: Carets and Dollars

Example

```
^fast

fast$
```

matched: faster, fastest, fastidious; breakfast, Belfast.

Quantifiers

Quantifiers provide a means of specifying how many times a RE should match.

Quantifier	Answers
(pattern)?	pattern is repeated zero or one time
(pattern)*	pattern is repeated zero or more times
(pattern)+	pattern is repeated one or more times
(pattern){m,n}	pattern is repeated from m to n times

Match One of More Times: The Plus Mark

The plus mark + essentially says to match the *preceding* regular expression at least once.

Word Boundaries

In order to match articles that occur at the beginning or end of a line as well as articles that occur in between, we use

Example

```
\b(a|an|the)\b
```

NB: backslash b will only match whitespace or non-alphanumeric characters.

Greedy and non-greedy

Greedy means that it will match as many repetitions as possible.

"*" Matches 0 or more (greedy) repetitions of the preceding RE.

"+" Matches 1 or more (greedy) repetitions of the preceding RE.

"?" Matches 0 or 1 (greedy) of the preceding RE.

"*?,+?,??" **Non-greedy versions of the previous three special characters.**

So this means, when the star, the plus, and the question mark appears alone, the computer executes it in a greedy way, but when any of these three character is followed by a Q mark, the computer executes it in a non-greedy way. It can also be used to "{m,n}?" to indicate the non-greedy version of {m,n}.

More FUN Exercises with NEMO!

Let's play with `nltk.app.nemo()`, a graphical interface for exploring REs.

RE in python

The `re` module contains several useful functions for working with regex.

Function	Description
compile(pattern[, flags])	Create a pattern object from a string with a regex
search(pattern, string[, flags])	Searches for `pattern` in `string`
match(pattern, string[, flags])	Matches `pattern` at the beginning of `string`
split(pattern, string[, maxsplit=0])	Returns a list of all occurrences of `pattern` in `string`
findall(pattern, string)	Returns a list of all occurrences of `pattern` in `string`
sub(pat, repl, string[, count=0])	Substitutes occurrences of `pat` in `string` with `repl`
escape(string)	Escape all regex characters in `string`

Exercise

Read Google's Python Class and do the part A exercise (with `nltk.re_show()` or your own function).

Cheatsheet

Table 3-3. Basic regular expression metacharacters, including wildcards, ranges, and closures

Operator	Behavior
`.`	Wildcard, matches any character
`^abc`	Matches some pattern *abc* at the start of a string
`abc$`	Matches some pattern *abc* at the end of a string
`[abc]`	Matches one of a set of characters
`[A-Z0-9]`	Matches one of a range of characters
`ed|ing|s`	Matches one of the specified strings (disjunction)
`*`	Zero or more of previous item, e.g., `a*`, `[a-z]*` (also known as *Kleene Closure*)
`+`	One or more of previous item, e.g., `a+`, `[a-z]+`
`?`	Zero or one of the previous item (i.e., optional), e.g., `a?`, `[a-z]?`
`{n}`	Exactly *n* repeats where *n* is a non-negative integer
`{n,}`	At least *n* repeats
`{,n}`	No more than *n* repeats
`{m,n}`	At least *m* and no more than *n* repeats
`a(b|c)+`	Parentheses that indicate the scope of the operators

Cheatsheet

Table 3-4. Regular expression symbols

Symbol	Function
\b	Word boundary (zero width)
\d	Any decimal digit (equivalent to [0-9])
\D	Any non-digit character (equivalent to [^0-9])
\s	Any whitespace character (equivalent to [\t\n\r\f\v]
\S	Any non-whitespace character (equivalent to [^ \t\n\r\f\v])
\w	Any alphanumeric character (equivalent to [a-zA-Z0-9_])
\W	Any non-alphanumeric character (equivalent to [^a-zA-Z0-9_])
\t	The tab character
\n	The newline character

Regular Expressions for Corpus Processing

1. Finding interesting patterns is a crucial work for corpus linguistics.
2. Regular expressions can be applied at different stages of corpus processing.
3. Combing with `re` and `NLTK` modules to get things easier.