

Python Course: Lecture 6

January 17, 2006

1 Text *qua* Language

- Consider a block of text: what are the meaningful elements? (Probably: words, punctuation marks, sentences, paragraphs)

Soon Starbuck returned with a letter in his hand. It was sorely tumbled, damp, and covered with a dull, spotted, green mould, in consequence of being kept in a dark locker of the cabin. Of such a letter, Death himself might well have been the post-boy.

"Can'st not read it?" cried Ahab. "Give it me, man. Aye, aye, it's but a dim scrawl;--what's this?" As he was studying it out, Starbuck took a long cutting-spade pole, and with his knife slightly split the end, to insert the letter there, and in that way, hand it to the boat, without its coming any closer to the ship.

Meantime, Ahab holding the letter, muttered, "Mr. Har--yes, Mr. Harry--(a woman's pinny hand,--the man's wife, I'll wager)--Aye--Mr. Harry Macey, Ship Jeroboam;--why it's Macey, and he's dead!"

"Poor fellow! poor fellow! and from his wife," sighed Mayhew; "but let me have it."

"Nay, keep it thyself," cried Gabriel to Ahab; "thou art soon going that way."

- What can you tell me about this text?
 - What language is it written in?
 - What is it about? (You could imagine an SAT-style quiz.)
 - What genre/style is it. (Literary text.)

- What century was it written in?
- What people/entities are involved?

How do you know all of this? How much of this knowledge could be imparted to a computer?

2 Text *qua* Text

- Linguists care about text. Computers store texts in strings. But extracting the linguistically relevant parts of a string can be tricky. This general task is known as *text processing*.
- Say I have this block of text in a string. How do I find the words?
- What about when punctuation abuts a word (“dead!”)?
- What about punctuation that belongs as part of a word (“Dr.”)?
- What about word fragments (“Har–”)?
- What about multicharacter punctuation (em-dash vs. hyphen)
- What about hyphenated words?
- Is “Poor” the same as “poor”? (Answer: it depends.)
- Note: I’m restricting the question to English for the time being—other writing systems will have other issues.
- The task of dividing a string up into smaller linguistically relevant pieces is called *tokenization*, and it can be surprisingly tricky.

Some terminology:

- Text consists of a string of *tokens*. These could be words, punctuation marks, whatever’s relevant.
- We may divide the tokens up into equivalence classes called *types*. For example in “to be or not to be” there are 2 “be” word tokens but just one “be” word type.
- We may specify different equivalence classes. For example, we may decide for a particular application that capitalization is not important, so the tokens “The” and “the” will both map to the same type “the”. This is called *text normalization*.
- Different corpora may have different spelling conventions, e.g. someone writes “cause” and someone else writes “because”.

- If we're doing a study of syntax, maybe we want to break contractions like “don't” up into syntactically relevant pieces “do” “n't”.

In real-world linguistic applications you spend a tremendous amount of time on these sorts of issues. They're tricky and no fun, but high-level languages like Python are the best tools for dealing with them.

In English text, the most important delimiter is whitespace—spaces, tabs, newlines. Note that even though the computer meticulously keeps tracks of each whitespace character in a string, consecutive ones usually all blend together into a single delimiter in the eyes of a human reader.

3 Regular Expressions

The standard way to deal with string-handling questions in any computer language is *regular expressions*.

- What if we want string equivalence that works in both the U.S. and Canada?

```
>>> "color" == "color"
True
>>> "color" == "colour"
False
```

- We could list all the alternate spellings, but this is redundant.

```
>>> "colour" in ["color", "colour"]
True
```

- We want to generalize the idea of string *equivalence* to string *pattern matching*. In the example here we want a way to say “color may optionally have a ‘u’ in it, but otherwise the spelling does not vary.”
- Do this with *regular expressions*. Regular expressions are a standard macro language for specifying string patterns. They are standard across many different platforms/computer languages. Python implements a standard version of the regular expression syntax.
- Regular expressions come in via the `re` library.

```
>>> import re
```

- A regular expression is a string that has special escape codes for specifying patterns. We use the `re` library to try and match it against normal strings. Here's an example:

```
>>> re.match('colou?r', 'color')
<_sre.SRE_Match object at 0x62250>
>>> re.match('colou?r', 'colour')
<_sre.SRE_Match object at 0x62b80>
>>> re.match('colou?r', 'shape')
```

- The first argument is a regular expression and the second is a string. The `match` command returns a `SRE_Match` object if there is a match and `None` if there is not.
- Most of the regular expression looks like text. The difference is the `?` character, which means “match one or zero of the preceding characters”.
- Character classes: `\d \D \s \S \w \W []`
- Example: phone numbers.

```
>>> pn = re.compile('\d{3}-\d{4}')
>>> pn.match('555-6868')
<_sre.SRE_Match object at 0xb734d870>
>>> pn.match('555- 6868')
>>> pn.match('555-a868')
>>>
```

- Use vowels as an example of a user-defined character class.
- Repeating characters: `* ? + {min,max}`
- Use parentheses for grouping.
- Linguistic example: match words with a CV syllable structure, optionally ending in a consonant.

```
>>> word = re.compile('([^\aeiou][aeiou])+[^\aeiou]?')
>>> word.match('banana')
<_sre.SRE_Match object at 0xb7326820>
>>> word.match('django')
>>>
```

- Escaping with backslash.
- Different commands: `match`, `search`, `findall`, `finditer`.
- `MatchObject` methods: `group`, `start`, `end`, `span`.
- Anchors: `^` and `$`.

- Dereferencing groups with the `group` method.
- The parenthesis operator in regular expressions is overloaded for both grouping and capturing. You can specify groups to be non-capturing.

```
>>> word = re.compile('(?:[^\aeiou][\aeiou])+[^\aeiou]?')
>>> word.match('banana').groups()
()
```

- You can name parts of the matched expression

```
>>> pn = re.compile('(P<prefix>\d{3})-(P<extension>\d{4})')
>>> pn.match('345-6878').group('prefix')
'345'
>>> pn.match('345-6878').group('extension')
```

- Changing strings: `split` and `sub`.
- Example, make a string into “Arabic” text.

```
>>> s = "English vowel information is actually somewhat redundant"
>>> v = re.compile('[\aeiou]+')
>>> v.sub('', s)
'Englsh vwl nfrmtn s ctllly smwht rdndnt'
```

- Compile versus direct execution.
- Example, enumerate sentences delimited by periods.

```
>>> srx = '(P<sentence>(?:\w+\s*)+\.)\s*'
>>> sent = re.compile(srx)
>>> text = "I ran away. Then I came home again."
>>> [m.group('sentence') for m in re.finditer(text)]
['I ran away.', 'Then I came home again.']
```

- Flags. There are many, but I’m just going to talk about `VERBOSE`.
- Example in `srxex.py`.
- Because regular expressions are like a miniature programming language, they can be involved and take some debugging. The best way to work is to build them incrementally and test them in the interactive mode.