Chapter 3. Processing Raw Text

Behrang QasemiZadeh

Outline

- Accessing text
    - From local files
    - From the web
- Regular Expressions
- Text Sectioning and Segmentation
    - Tokenization
    - Stemming
- Producing formatted outputs

Accessing a text file from the web

```
>>> from urllib import urlopen
>>> url = "http://www.gutenberg.org/files/2554/2554.txt"
>>> raw = urlopen(url).read()
>>> type(raw)
<type 'str'>
>>> len(raw)
1176831
>>> raw[:75]
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'
```

Accessing a text file from the web

Source text file on the web

```
>>> from urllib import urlopen
>>> url = "http://www.gutenberg.org/files/2554/2554.txt"
>>> raw = urlopen(url).read()
>>> type(raw)
<type 'str'>
>>> len(raw)
1176831
>>> raw[:75]
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'
```

Accessing a text file from the web

```
>>> from urllib import urlopen
>>> url = "http://www.gutenberg.org/files/2554/2554.txt"
>>> raw = urlopen(url).read()
>>> type(raw)
<type 'str'>
>>> len(raw)
1176831
>>> raw[:75]
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'
```

Tell me the type of accessed material!

Accessing a text file from the web

```
>>> from urllib import urlopen
>>> url = "http://www.gutenberg.org/files/2554/2554.txt"
>>> raw = urlopen(url).read()
>>> type(raw)
<type 'str'>
>>> len(raw)
1176831
>>> raw[:75]
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'
```

Give me the length of text

Accessing a text file from the web

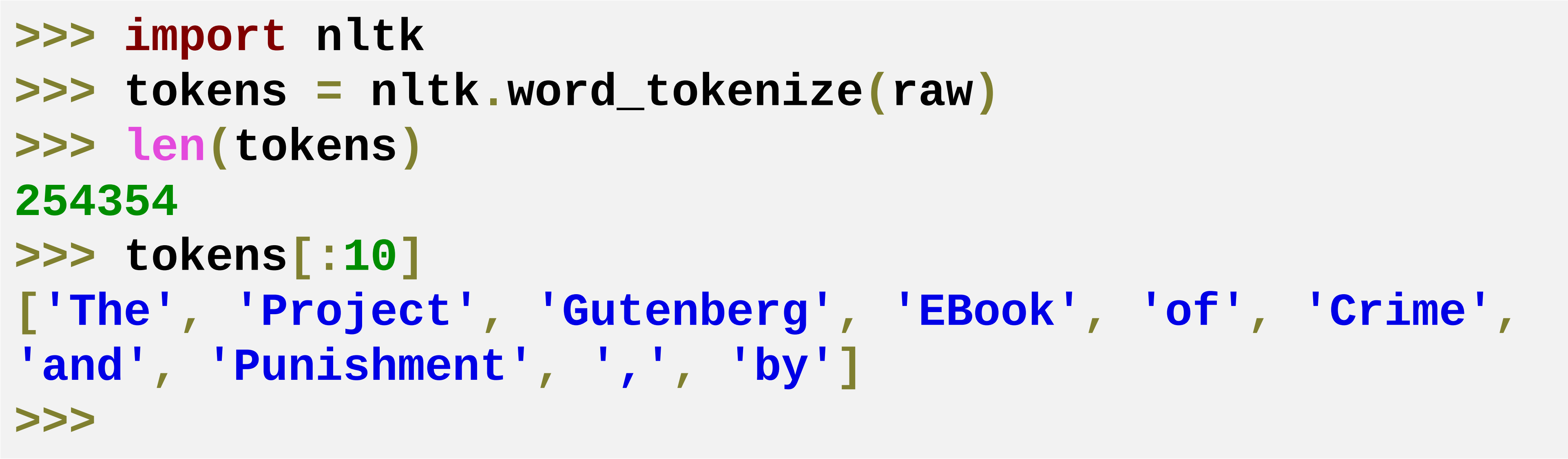
```
>>> from urllib import urlopen
>>> url = "http://www.gutenberg.org/files/2554/2554.txt"
>>> raw = urlopen(url).read()
>>> type(raw)
<type 'str'>
>>> len(raw)
1176831
>>> raw[:75]
'The Project Gutenberg EBook of Crime and Punishment, by Fyodor Dostoevsky\r\n'
```

Give me the first 75 characters!

Accessing a text file from hard drive

- Let’s discuss assignment 1 and 2!

Text Tokenization



```
>>> import nltk
>>> tokens = nltk.word_tokenize(raw)
>>> len(tokens)
254354
>>> tokens[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime',
'and', 'Punishment', ',', 'by']
>>>
```

Text Tokenization

Import NLTK and use tokenization function

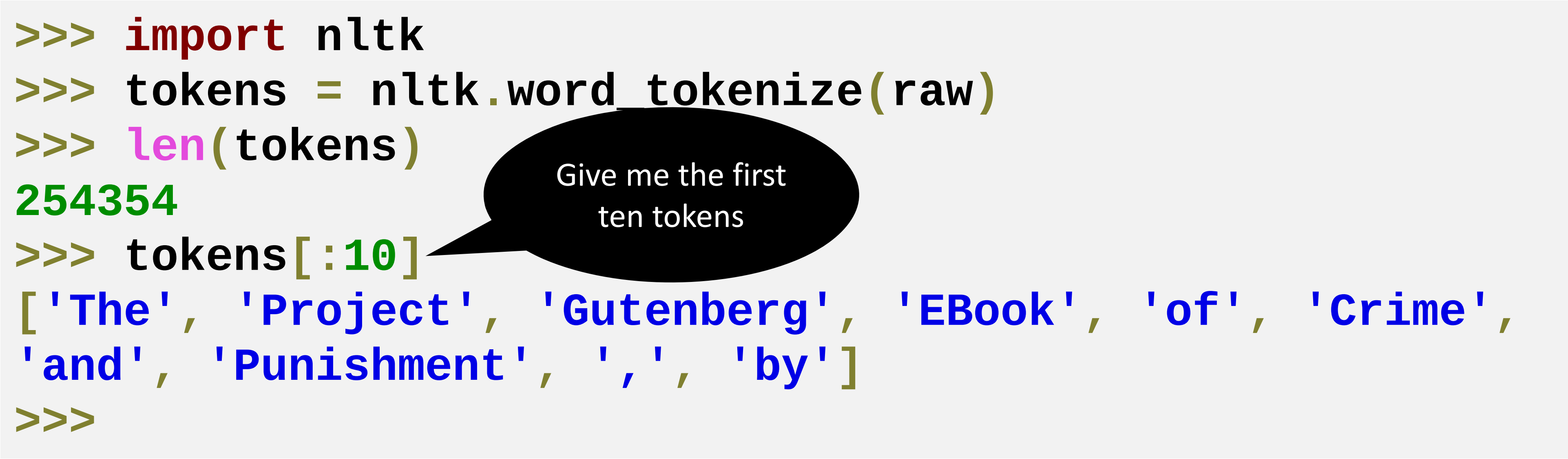
```
>>> import nltk
>>> tokens = nltk.word_tokenize(raw)
>>> len(tokens)
254354
>>> tokens[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime',
'and', 'Punishment', ',', 'by']
>>>
```

Text Tokenization

```
>>> import nltk
>>> tokens = nltk.word_tokenize(raw)
>>> len(tokens)
254354
>>> tokens[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime',
'and', 'Punishment', ',', 'by']
>>>
```

Give me number of tokens

Text Tokenization



```
>>> import nltk
>>> tokens = nltk.word_tokenize(raw)
>>> len(tokens)
254354
>>> tokens[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Crime',
'and', 'Punishment', ',', 'by']
>>>
```

Give me the first ten tokens

Quiz

- What is the `type()` of tokens in the following code?

```
>>> import nltk
>>> tokens = nltk.word_tokenize(raw)
```

Quiz

- What is the `type()` of tokens in the following code?

```
>>> import nltk
>>> tokens = nltk.word_tokenize(raw)
```

LIST!

Simple Text Segmentation Using `find()`

- Text Segmentation might be required for reducing noise.
- A raw text file may contain a header or a footer, e.g. in the beginning of a text file we may see:
    - copyright notice
    - project information
    - etc.

```
>>> raw[:75]
'The Project Gutenberg EBook of Crime and Punishment,
by Fyodor Dostoevsky\r\n'
```

Simple Text Segmentation Using `find()`

- Sometimes, a manual inspection can help the identification of text segments, e.g. using unique strings that mark beginning and end of text files.
- `find()`and `rfind()` can be used in these cases:

```
>>> raw.find("PART I")
5338
>>> raw.rfind("End of Project Gutenberg's Crime")
1157743
>>> raw[5303: 5471]
'\n\r\n\r\nCRIME AND PUNISHMENT\r\n\r\n\r\n\r\n\r\nPART
I\r\n\r\n\r\n\r\nCHAPTER I\r\n\r\nOn an exceptionally hot evening early in
July a young man came out of\r\nthe garret in which he lodged in S.'
>>>
```

Dealing with HTML

- HTML documents are frequent on the web:

```
>>> from urllib import urlopen
>>> url = “http://news.bbc.co.uk/2/hi/health/2284783.stm”
>>> html = urlopen(url).read()
>>> html[:20]
'<!doctype html public'
>>>
```

- To extract raw text from a HTML file, we must first get rid of HTML mark-ups.

Dealing with HTML – Using Beautiful Soup

- **`nltk.clean_html()`**used to be used to strip HTML tags from your fetched string.
  - The function is dropped since better alternatives are available.
- Beautiful Soup is a module that provides functionalities for removing HTML tags.
- To install Beautiful Soup:
  - `$apt-get install python-bs4`
  - `$pip install beautifulsoup4`
  - you can download the Beautiful Soup 4 source tarball and install it with setup.py

    ```
    $ python setup.py install
    ```
- See [**`http://www.crummy.com/software/BeautifulSoup/bs4/doc/`**](http://www.crummy.com/software/BeautifulSoup/bs4/doc/) for documentation.

Beautiful Soup Example

```
>>> from urllib import urlopen
>>> url = “http://news.bbc.co.uk/2/hi/health/2284783.stm”
>>> html = urlopen(url).read()
>>> html[:60]
'<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN‘
>>> soup = BeautifulSoup(html)
>>> clean_text = soup.get_text()
>>> clean_text[:60]
u"\n\n\nBBC NEWS | Health | Blondes 'to die out in 200
years'\n\n\n\n"
>>>
```

Beautiful Soup Example

```
>>> from urllib import urlopen
>>> url = “http://news.bbc.co.uk/2/hi/health/2284783.stm”
>>> html = urlopen(url).read()
>>> html[:60]
'<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN‘
>>> soup = BeautifulSoup(html)
>>> clean_text = soup.get_text()
>>> clean_text[:60]
u"\n\n\nBBC NEWS | Health | Blondes 'to die out in 200
years'\n\n\n\n"
>>>
```

Compare the outputs

Processing RSS Feeds

- RSS feeds can be also accessed using **`feedparser`** module.

```
>>> import feedparser
>>> llog = feedparser.parse(
            "http://languagelog.ldc.upenn.edu/nll/?feed=atom")
>>> llog['feed']['title']
u'Language Log'
>>> len(llog.entries)
15
>>> post = llog.entries[2]
>>> post.title
u'A child&#039;s substitution of Pinyin (Romanization) for characters'
>>> content = post.content[0].value
>>> content[:70]
u'<p>The following diary entry by an elementary school student is making'
>>>
```

Processing RSS Feeds

- RSS feeds can be also accessed using **`feedparser`** module.

```
>>> import feedparser
>>> llog = feedparser.parse(
            "http://languagelog.ldc.upenn.edu/nll/?feed=atom")
>>> llog['feed']['title']
u'Language Log'
>>> len(llog.entries)
15
>>> post = llog.entries[2]
>>> post.title
u'A child&#039;s substitution of Pinyin (Romanization) for characters'
>>> content = post.content[0].value
>>> content[:70]
u'<p>The following diary entry by an elementary school student is making'
>>>
```

Locate the feed and fetch

Processing RSS Feeds

- RSS feeds can be also accessed using **`feedparser`** module.

```
>>> import feedparser
>>> llog = feedparser.parse(
             "http://languagelog.ldc.upenn.edu/nll/?feed=atom")
>>> llog['feed']['title']
u'Language Log'
>>> len(llog.entries)
15
>>> post = llog.entries[2]
>>> post.title
u'A child&#039;s substitution of Pinyin (Romanization) for characters'
>>> content = post.content[0].value
>>> content[:70]
u'<p>The following diary entry by an elementary school student is making'
>>>
```

Work with content!

Processing RSS Feeds: accessing news

```
>>> import feedparser
>>> rssbbcnews =
feedparser.parse("http://feeds.bbci.co.uk/news/world/rss.xml")
>>> rssbbcnews ['feed']['title']
u'BBC News - World'
>>> len(rssbbcnews.entries)
53
>>> postbbc = rssbbcnews.entries[1]
>>> postbbc.title
u'EU court backs migrant benefit curbs
>>> postbbc.description
u"The European Court of Justice backs curbs on unemployed migrants' access to
certain benefits, setting a legal precedent for all EU member states."
>>>
```

Quiz: Processing RSS Feeds

In the previous example, what is the output for

```
bbcpost.content[0].value
```

and Why?

Quiz: Processing RSS Feeds

In the previous example, what is the output for

```
bbcpost.content[0].value
```

and Why?

RSS structure
is important!

Capturing user Input

- Python function **`raw_input()`** can be used to prompt the user to type a line of input:

```
>>> text = raw_input("how are you? ")
how are you? I am fine!
>>> print "You typed", len(nltk.word_tokenize(text)),\
      "words: " , text
You typed 4 words: I am fine!
>>>
```

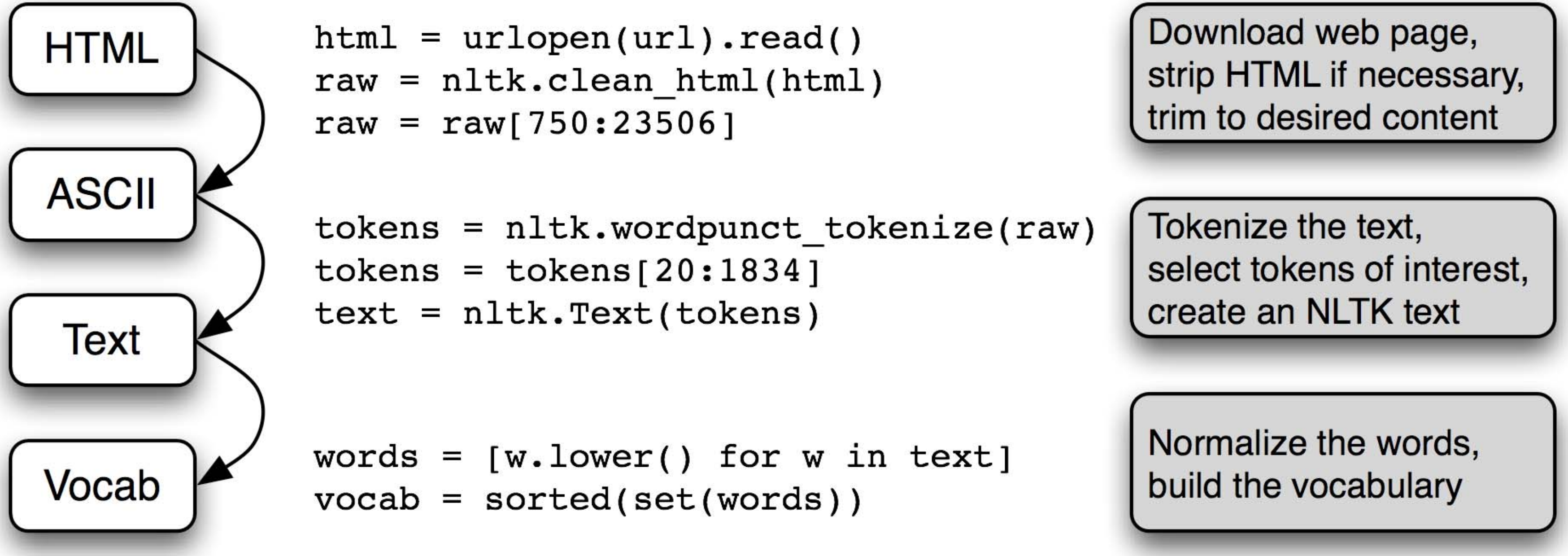
Additional Text Sources

- There are a number of other sources to access raw text strings.
- There are often specialized APIs that let you access text from different platforms:
  - Accessing text from XML files
  - Accessing text from databases
  - Accessing text from social networks
  - Accessing text from MS Word and PDF files
  - Etc.

Quiz – Programming Exercise

- Implement codes for an *NLP Pipeline* that
  1. Fetches the text from an HTML file on the web
  2. Strip off HTML tags and get clean text strings
  3. Convert the text into vocab/lexicon, **i.e. a list of sorted words**

Quiz – Programming Exercise



```
html = urlopen(url).read()
raw = nltk.clean_html(html)
raw = raw[750:23506]

tokens = nltk.wordpunct_tokenize(raw)
tokens = tokens[20:1834]
text = nltk.Text(tokens)

words = [w.lower() for w in text]
vocab = sorted(set(words))
```

Download web page, strip HTML if necessary, trim to desired content

Tokenize the text, select tokens of interest, create an NLTK text

Normalize the words, build the vocabulary

Processing of Strings (Review)

- Strings are marked by ‘ or “
  - If a string is too long, you can break it down by using parentheses or \

```
>>> couplet = "Shall I compare thee to a Summer's day?"
>>> couplet
"Shall I compare thee to a Summer's day?"
>>> couplet = "Shall I compare thee to a Summer's day?"\
       "Though are '''"
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = "Shall I compare thee to a Summer's day?"\
       'Though are \'\'\''
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = ("shall I...“
       "Though are...")
>>> couplet
'shall I...Though are...'
>>> couplet = """shall I... Though are..."""
>>> couplet 'shall I...\nThough are...'
```

Processing of Strings (Review)

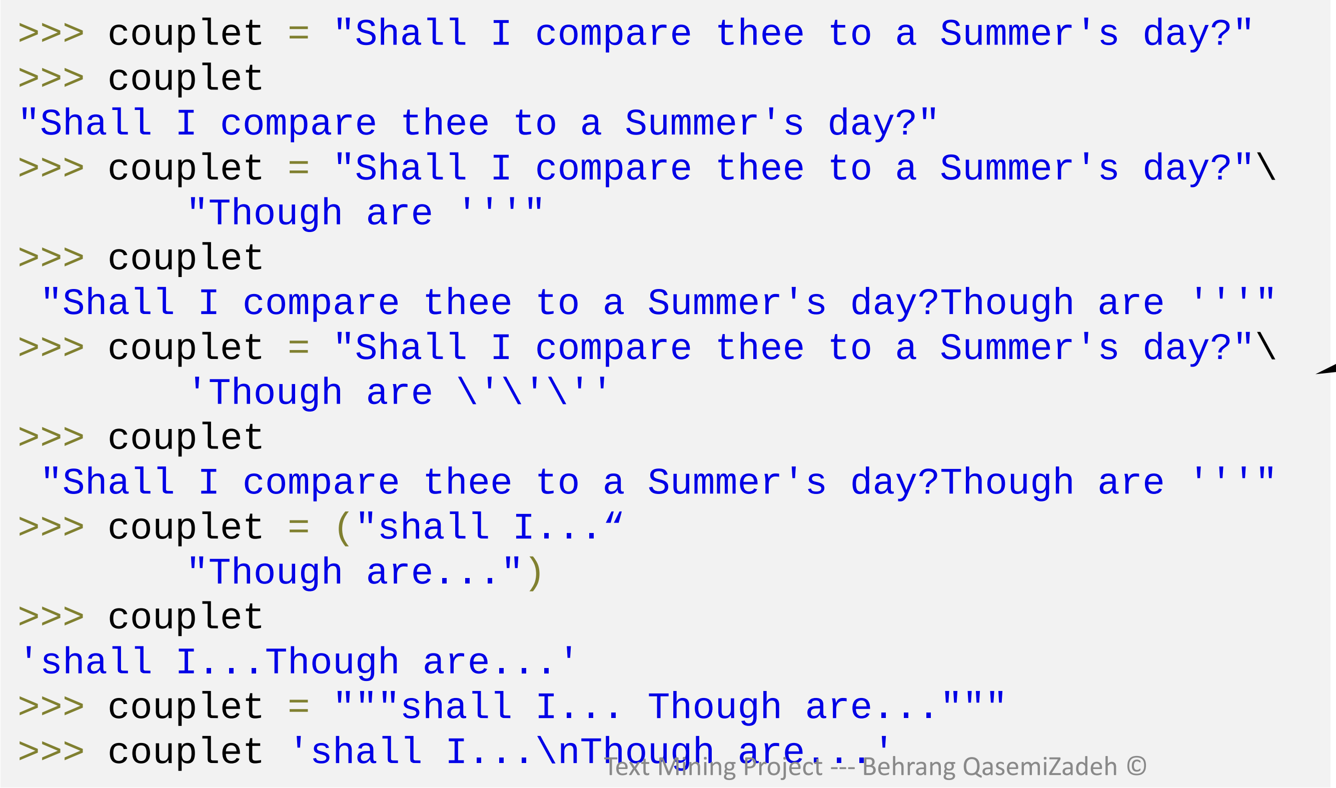
- Strings are marked by ‘ or “
  - If a string is too long, you can break it down by using parentheses or \

```
>>> couplet = "Shall I compare thee to a Summer's day?"
>>> couplet
"Shall I compare thee to a Summer's day?"
>>> couplet = "Shall I compare thee to a Summer's day?"\
        "Though are '''"
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = "Shall I compare thee to a Summer's day?"\
        'Though are \'\'\''
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = ("shall I...“
        "Though are...")
>>> couplet
'shall I...Though are...'
>>> couplet = """shall I... Though are..."""
>>> couplet 'shall I...\nThough are...'
```

Pay attention to the use of \ for breaking long lines

Processing of Strings (Review)

- Strings are marked by ‘ or “
  - If a string is too long, you can break it down by using parentheses or \



```
>>> couplet = "Shall I compare thee to a Summer's day?"
>>> couplet
"Shall I compare thee to a Summer's day?"
>>> couplet = "Shall I compare thee to a Summer's day?"\
       "Though are '''"
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = "Shall I compare thee to a Summer's day?"\
       'Though are \'\'\''
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = ("shall I...“
       "Though are...")
>>> couplet
'shall I...Though are...'
>>> couplet = """shall I... Though are..."""
>>> couplet 'shall I...\nThough are...'
```

Pay attention to the scape character uses with single quotes.

Processing of Strings (Review)

- Strings are marked by ‘ or “
  - If a string is too long, you can break it down by using parentheses or \

```
>>> couplet = "Shall I compare thee to a Summer's day?"
>>> couplet
"Shall I compare thee to a Summer's day?"
>>> couplet = "Shall I compare thee to a Summer's day?"\
       "Though are '''"
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = "Shall I compare thee to a Summer's day?"\
       'Though are \'\'\''
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = ("shall I...“
       "Though are...")
>>> couplet
'shall I...Though are...'
>>> couplet = """shall I... Though are..."""
>>> couplet 'shall I...\nThough are...'
```

Pay attention to the use of parentheses for breaking long strings

Processing of Strings (Review)

- Strings are marked by ‘ or “
  - If a string is too long, you can break it down by using parentheses or \

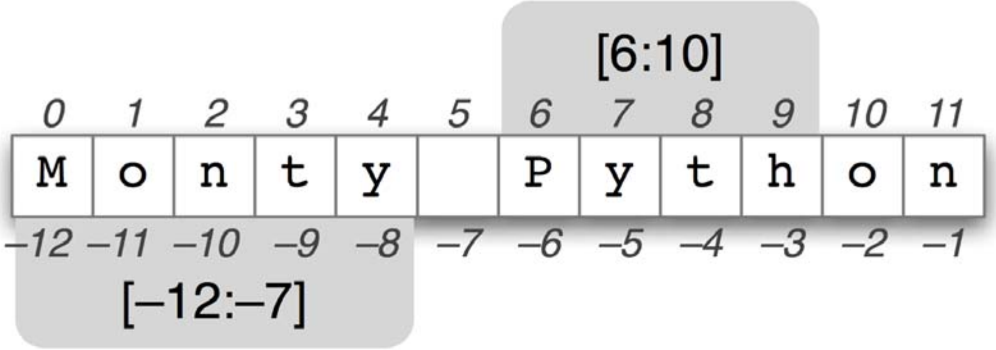
```
>>> couplet = "Shall I compare thee to a Summer's day?"
>>> couplet
"Shall I compare thee to a Summer's day?"
>>> couplet = "Shall I compare thee to a Summer's day?"\
       "Though are '''"
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = "Shall I compare thee to a Summer's day?"\
       'Though are \'\'\''
>>> couplet
 "Shall I compare thee to a Summer's day?Though are '''"
>>> couplet = ("shall I...“
       "Though are...")
>>> couplet
'shall I...Though are...'
>>> couplet = """shall I... Though are..."""
>>> couplet 'shall I...\nThough are...'
```

Pay attention to the use of triple double quotes and the inserted \n in the string

Important String Operations (review)

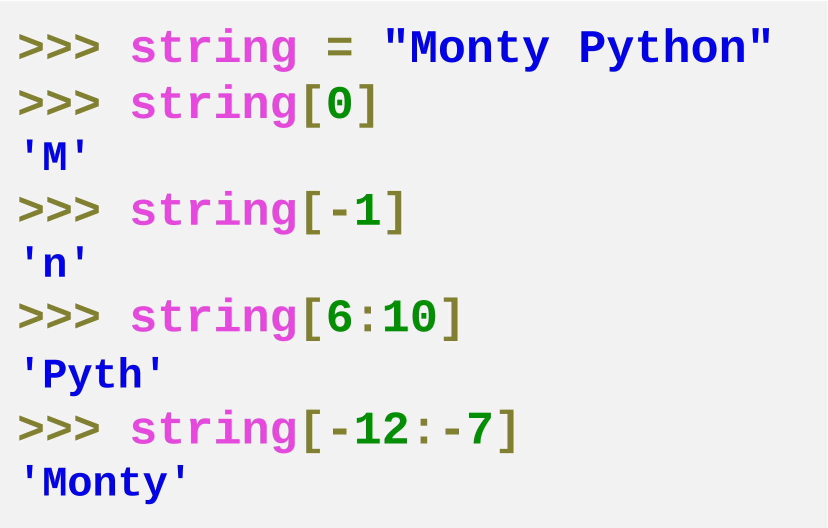
- Accessing individual characters and substrings using “string slicing”

```
>>> string = "Monty Python"
>>> string[0]
'M'
>>> string[-1]
'n'
>>> string[6:10]
'Pyth'
>>> string[-12:-7]
'Monty'
```

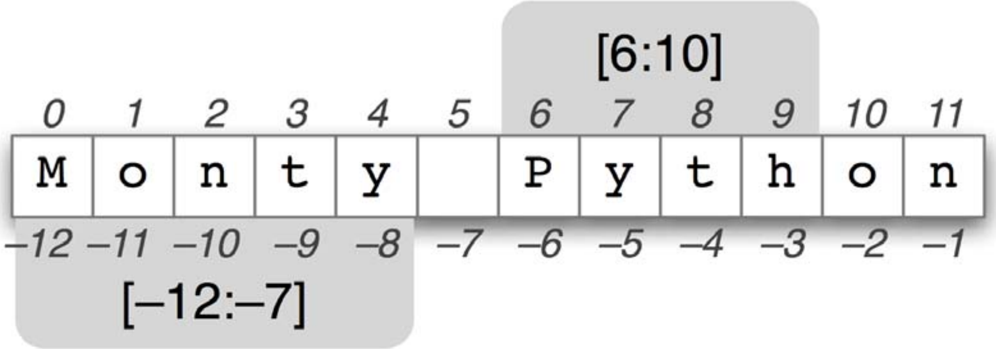


Important String Operations (review)

- Accessing individual characters and substrings using “string slicing”



```
>>> string = "Monty Python"
>>> string[0]
'M'
>>> string[-1]
'n'
>>> string[6:10]
'Pyth'
>>> string[-12:-7]
'Monty'
```



Find the position of substrings using `find()` (Example given in previous slides)

Further String Operations

Method	Functionality
`s.find(t)`	Index of first instance of string t inside s (-1 if not found)
`s.rfind(t)`	Index of last instance of string t inside s (-1 if not found)
`s.index(t)`	Like `s.find(t)`, except it raises ValueError if not found
`s.rindex(t)`	Like `s.rfind(t)`, except it raises ValueError if not found
`s.join(text)`	Combine the words of the text into a string using s as the glue
`s.split(t)`	Split s into a list wherever a t is found (whitespace by default)
`s.splitlines()`	Split s into a list of strings, one per line
`s.lower()`	A lowercased version of the string s
`s.upper()`	An uppercased version of the string s
`s.titlecase()`	A title-cased version of the string s
`s.strip()`	A copy of s without leading or trailing whitespace
`s.replace(t, u)`	Replace instances of t with u inside s

Quiz

- In python, what are the similarities and differences between lists and strings?

Quiz

- In python, what are the similarities and differences between lists and strings?
  - Similarities:
    - Both represent sequential data.
    - Both list and string can be manipulated by indexing and slicing.
  - Differences:
    - Lists may represent data at different level of granularity, e.g. lists of lists, but strings only represent sequence of characters (i.e. fixed granularity)
    - Lists are **mutable** but strings are **immutable**

Quiz

- In python, what are the similarities and differences between lists and strings?
  - Similarities:
    - Both represent sequential data.
    - Both list and string can be manipula
  - Differences:
    - Lists may represent data at diffe lists of lists, but strings only repre (i.e. fixed granularity)
    - Lists are **mutable** but strings are **imm**

Experiment:

Define a string (e.g. `textString`) and a list (e.g. `listString`) and assign values to them, then try to change the first element of each using the index, e.g. `textString[0] = “A”` or `listString[0] = “L1”`

Text Encoding and Unicode

- ASCII is a character encoding system that only supports only 128 different characters (for 7-bit encoding system or 255 for single byte):
  - Sufficient for English text (or when we only deal with 128characters)
  - Insufficient for many other languages, e.g. how to represent Arabic character ى or ل?
  - What if we want to deal with text in Chinese, English and Arabic at the same time?
- Unicode supports over a million characters:
  - A single **character set** that included every reasonable writing system
  - Each character is assigned a number, called a **code point**.
  - Code points are four digit hexadecimal numbers (in Python presented as \u*XXXX).*

Unicode: characters not glyphs

- In Unicode, characters are abstract entities that can have one or more **glyphs (that is the written shape)**.
  - For example in Arabic writing system a character can have 4 different glyphs:

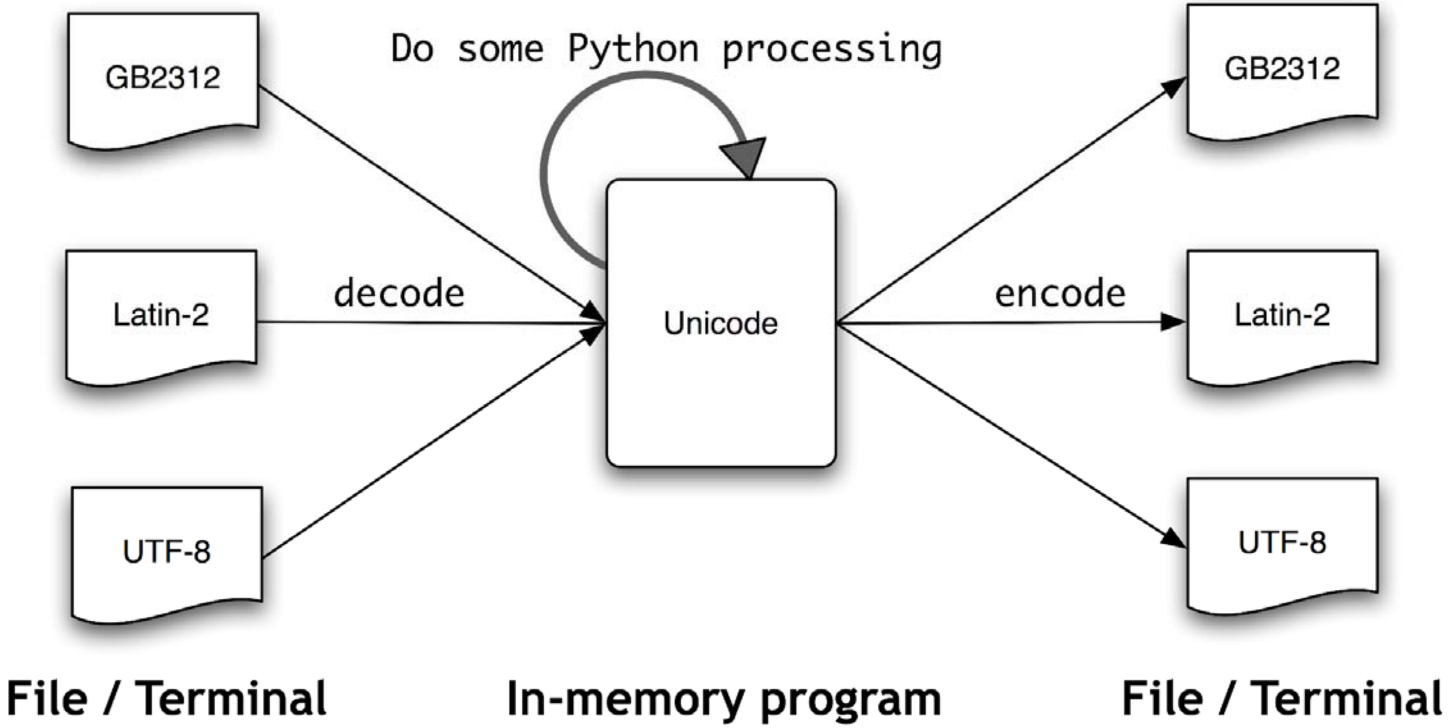
    For single character */ye/:* ی ـی ـیـ یـ

- A font system and additional algorithms take care after proper representation of character codes.

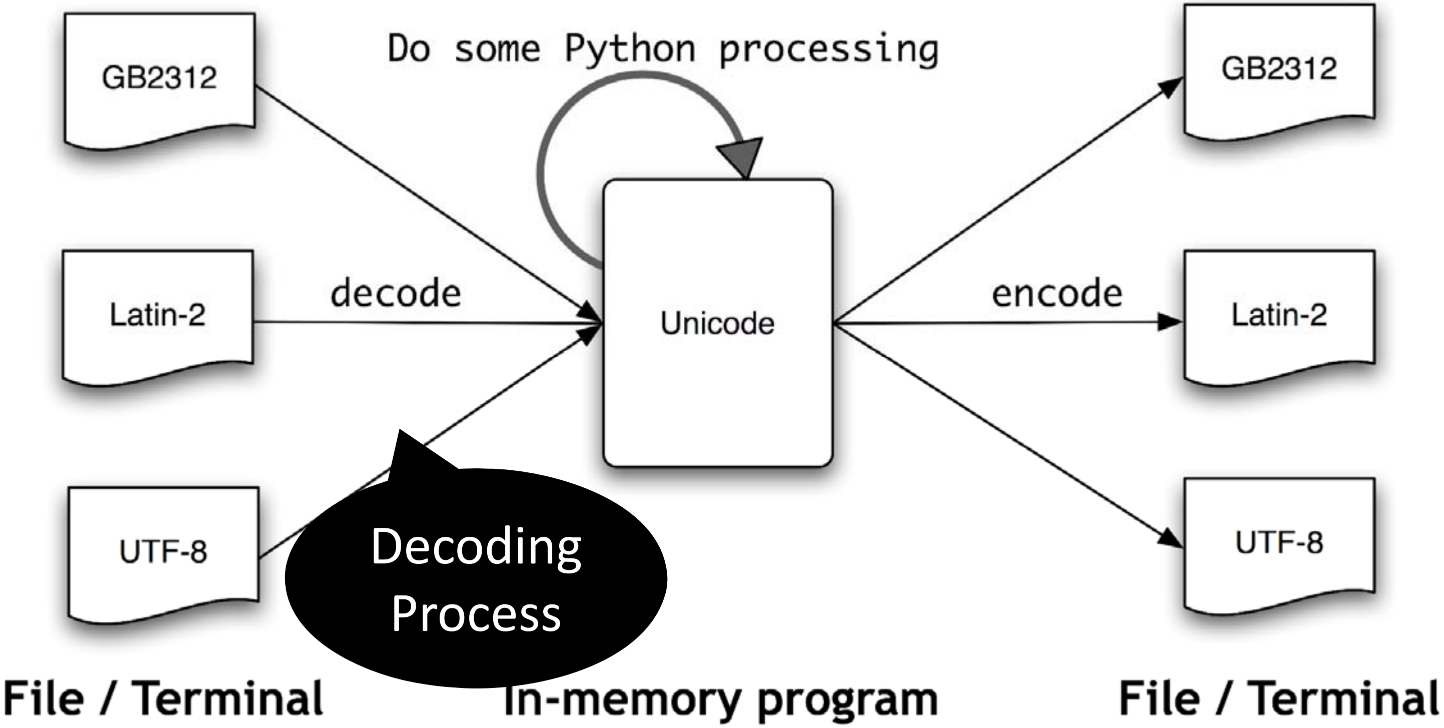
Unicode, ASCII, UTF-8 and other encodings

- ASCII can only represent a subset of Unicode characters.
- UTF-8 (amongst other encodings) uses multiple bytes and can represent the full range of Unicode characters.
  - Why UTF-8?

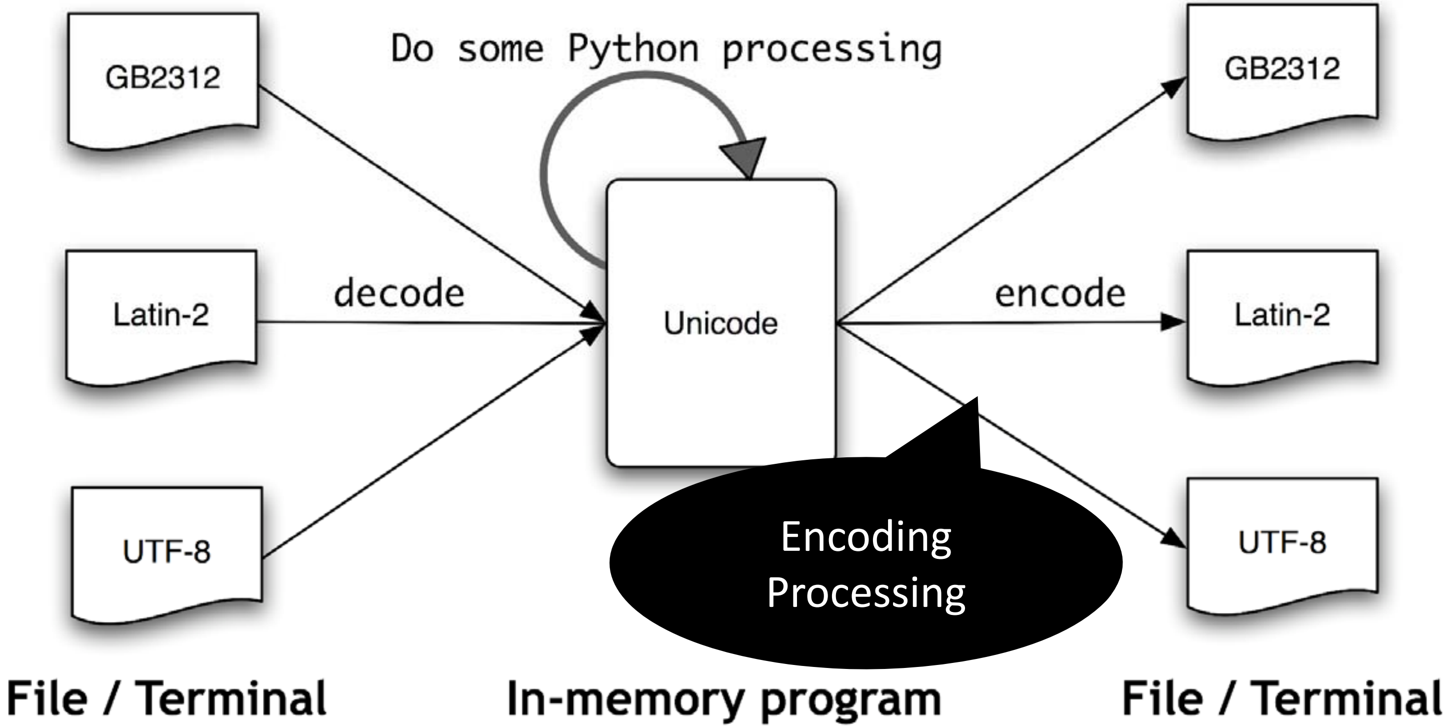
Unicode, ASCII, UTF-8 and other encodings



Unicode, ASCII, UTF-8 and other encodings



Unicode, ASCII, UTF-8 and other encodings



Encoding Processing

Extracting Encoded Text from Files

- The Python **codecs** module provides functions to read encoded data into Unicode strings.
  - Encoding can be set as a parameter in the codecs.open() function when the file being read or written:

```
>>> import codecs
>>> f = codecs.open(path, encoding='latin2')
```

- See [*http://docs.python.org/lib/standard-encodings.html*](http://docs.python.org/lib/standard-encodings.html) for the list of permitted encodings.

Exercise/Quiz

- Create a UTF-8 file using a text editor.
  - Read the file using ASCII encoding
  - Read the file using UTF-8 encoding
  - Compare the outputs
  - Convert the encoding of the file into Latin2
    - We can write Unicode-encoded data to a file using

```
f = codecs.open(path, 'w', encoding='latin2')
```

Extracting Encoded Text from Files

- Other methods you may want to know:
  - **`text.encode(‘unicode_escape’)`**: converts all non-ASCII characters in `text` into their \u*XXXX* representations.
  - **`u’\XXXX’`**: use to specify Unicode string literals.
  - **`ord(X):`** the integer ordinal of a character.
  - **`repr():`** outputs the UTF-8 escape sequences (of the form \x*XX*) rather than trying to render the glyphs.
  - Also see functions in the module **`unicodedata.`**

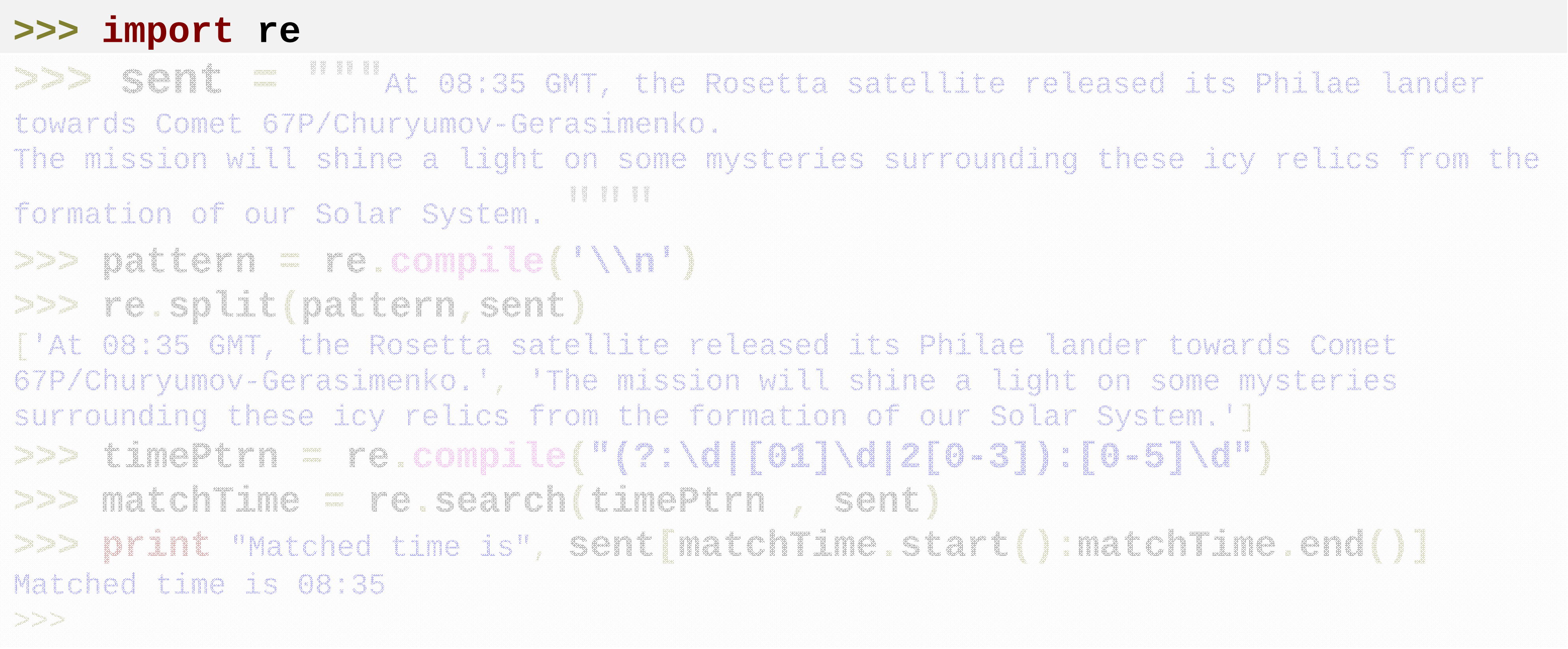
Further reading on encoding

- Must read: https://docs.python.org/2/howto/unicode.html
- Intro to character sets: http://www.cs.tut.fi/~jkorpela/chars.html
- Official Unicode site: http://www.unicode.org
- Also read http://www.joelonsoftware.com/articles/Unicode.html
- Python Unicode Objects: http://effbot.org/zone/unicode-objects.htm
- A tutorial: http://www.unicode.org/standard/tutorial-info.html

Using Regular Expressions in Python

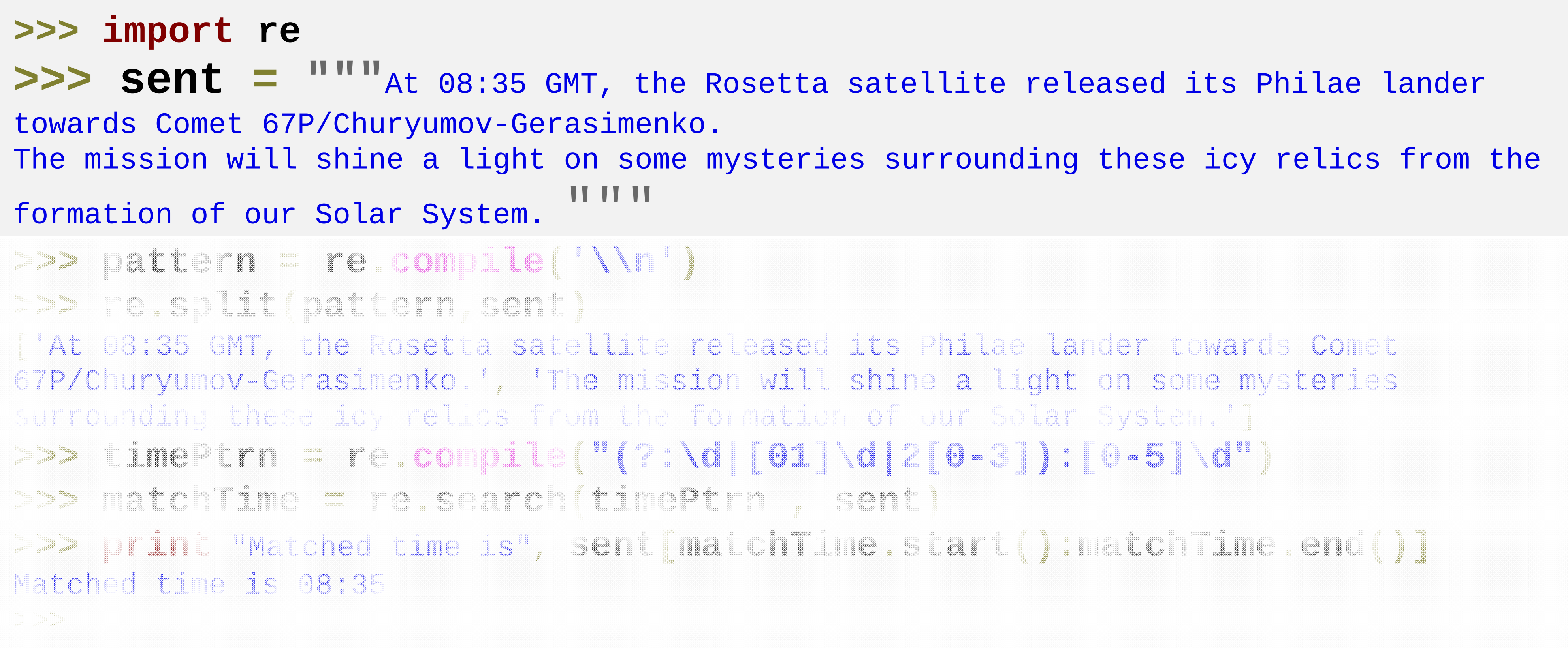
- Regular expressions give us a powerful and flexible method to describing character patterns that we are interested in.

Using Regular Expressions in Python



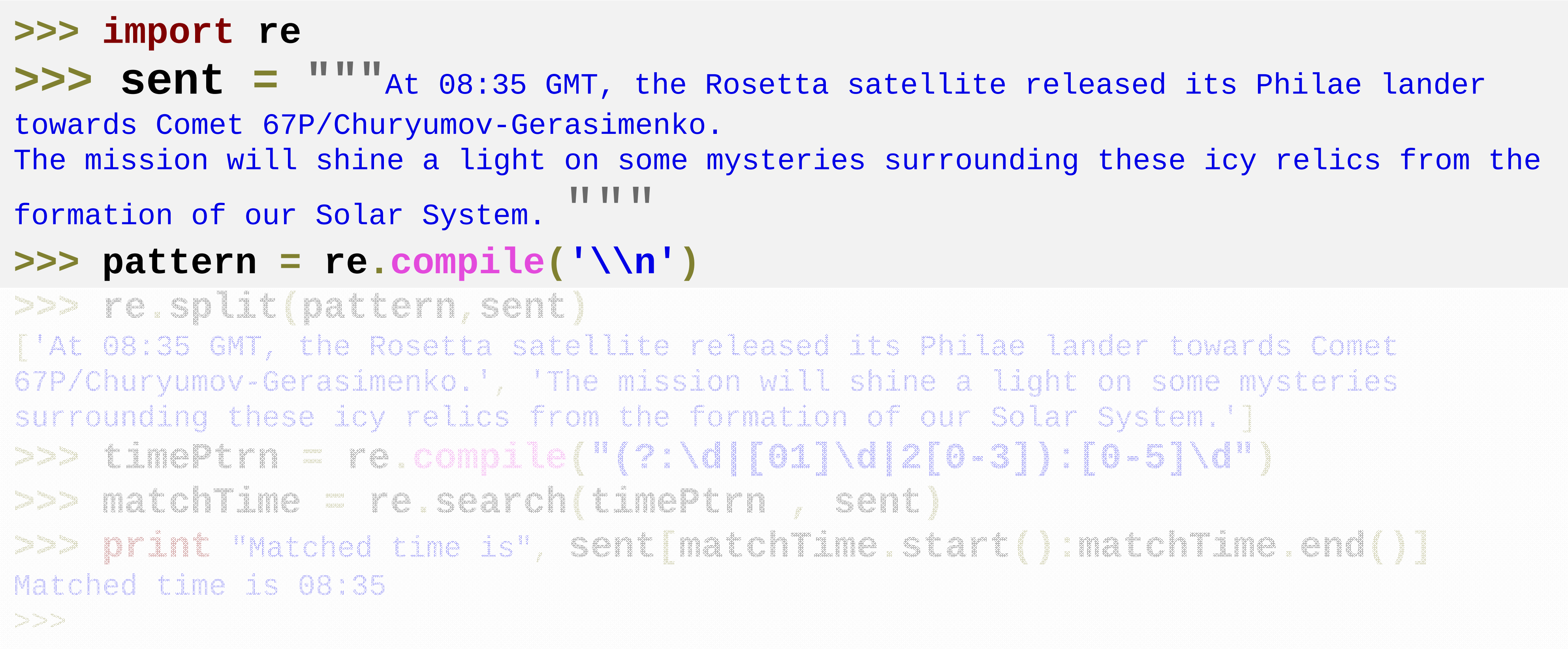
```
>>> import re
>>> sent = """At 08:35 GMT, the Rosetta satellite released its Philae lander
towards Comet 67P/Churyumov-Gerasimenko.
The mission will shine a light on some mysteries surrounding these icy relics from the
formation of our Solar System."""
>>> pattern = re.compile('\\n')
>>> re.split(pattern,sent)
['At 08:35 GMT, the Rosetta satellite released its Philae lander towards Comet
67P/Churyumov-Gerasimenko.', 'The mission will shine a light on some mysteries
surrounding these icy relics from the formation of our Solar System.']
>>> timePtrn = re.compile("(?:\d|[01]\d|2[0-3]):[0-5]\d")
>>> matchTime = re.search(timePtrn , sent)
>>> print "Matched time is", sent[matchTime.start():matchTime.end()]
Matched time is 08:35
>>>
```

Using Regular Expressions in Python



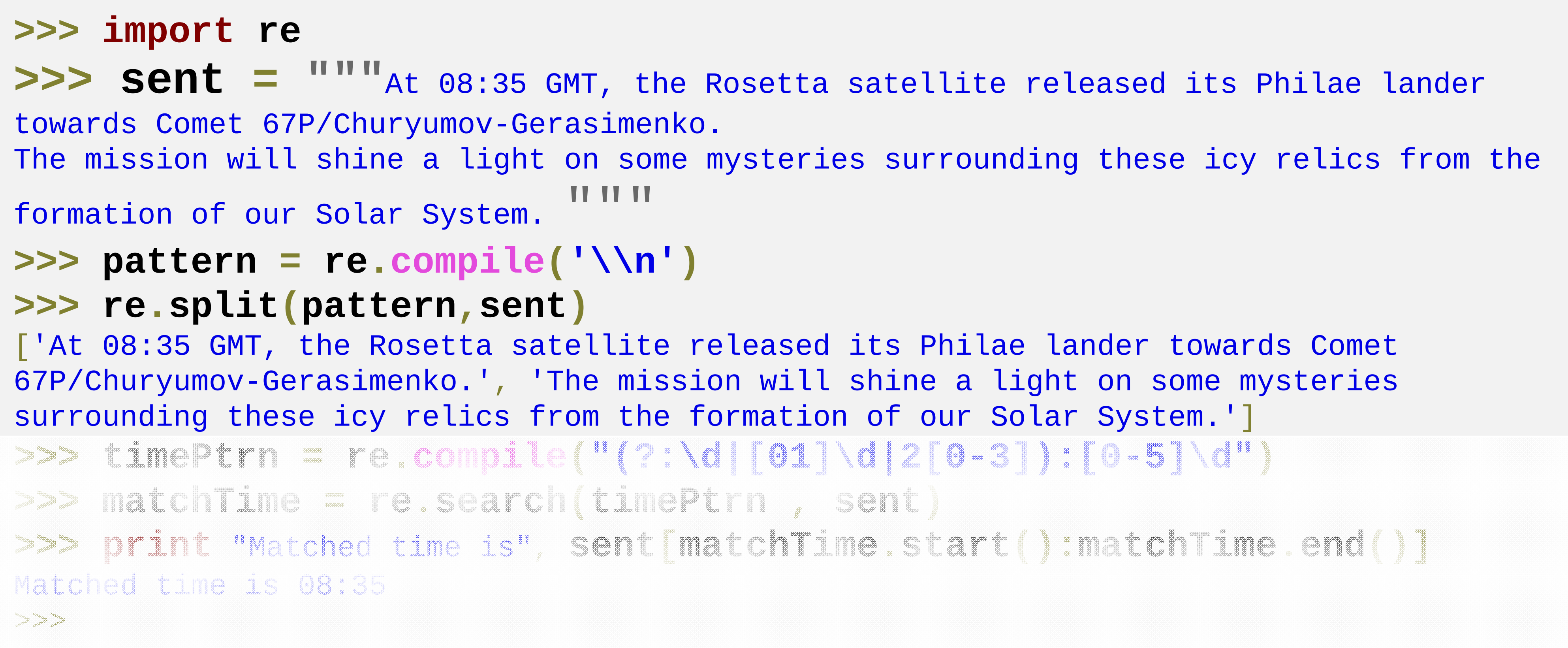
```
>>> import re
>>> sent = """At 08:35 GMT, the Rosetta satellite released its Philae lander
towards Comet 67P/Churyumov-Gerasimenko.
The mission will shine a light on some mysteries surrounding these icy relics from the
formation of our Solar System. """
>>> pattern = re.compile('\\n')
>>> re.split(pattern,sent)
['At 08:35 GMT, the Rosetta satellite released its Philae lander towards Comet
67P/Churyumov-Gerasimenko.', 'The mission will shine a light on some mysteries
surrounding these icy relics from the formation of our Solar System.']
>>> timePtrn = re.compile("(?:\d|[01]\d|2[0-3]):[0-5]\d")
>>> matchTime = re.search(timePtrn , sent)
>>> print "Matched time is", sent[matchTime.start():matchTime.end()]
Matched time is 08:35
>>>
```

Using Regular Expressions in Python



```
>>> import re
>>> sent = """At 08:35 GMT, the Rosetta satellite released its Philae lander
towards Comet 67P/Churyumov-Gerasimenko.
The mission will shine a light on some mysteries surrounding these icy relics from the
formation of our Solar System. """
>>> pattern = re.compile('\\n')
>>> re.split(pattern,sent)
['At 08:35 GMT, the Rosetta satellite released its Philae lander towards Comet
67P/Churyumov-Gerasimenko.', 'The mission will shine a light on some mysteries
surrounding these icy relics from the formation of our Solar System.']
>>> timePtrn = re.compile("(?:\d|[01]\d|2[0-3]):[0-5]\d")
>>> matchTime = re.search(timePtrn , sent)
>>> print "Matched time is", sent[matchTime.start():matchTime.end()]
Matched time is 08:35
>>>
```

Using Regular Expressions in Python



```
>>> import re
>>> sent = """At 08:35 GMT, the Rosetta satellite released its Philae lander
towards Comet 67P/Churyumov-Gerasimenko.
The mission will shine a light on some mysteries surrounding these icy relics from the
formation of our Solar System. """
>>> pattern = re.compile('\\n')
>>> re.split(pattern,sent)
['At 08:35 GMT, the Rosetta satellite released its Philae lander towards Comet
67P/Churyumov-Gerasimenko.', 'The mission will shine a light on some mysteries
surrounding these icy relics from the formation of our Solar System.']
>>> timePtrn = re.compile("(?:\d|[01]\d|2[0-3]):[0-5]\d")
>>> matchTime = re.search(timePtrn , sent)
>>> print "Matched time is", sent[matchTime.start():matchTime.end()]
Matched time is 08:35
>>>
```

Using Regular Expressions in Python

```
>>> import re
>>> sent = """At 08:35 GMT, the Rosetta satellite released its Philae lander
towards Comet 67P/Churyumov-Gerasimenko.
The mission will shine a light on some mysteries surrounding these icy relics from the
formation of our Solar System. """
>>> pattern = re.compile('\\n')
>>> re.split(pattern,sent)
['At 08:35 GMT, the Rosetta satellite released its Philae lander towards Comet
67P/Churyumov-Gerasimenko.', 'The mission will shine a light on some mysteries
surrounding these icy relics from the formation of our Solar System.']
>>> timePtrn = re.compile("(?:\d|[01]\d|2[0-3]):[0-5]\d")
>>> matchTime = re.search(timePtrn , sent)
>>> print "Matched time is", sent[matchTime.start():matchTime.end()]
Matched time is 08:35
>>>
```

Using Regular Expressions in Python

- Regular Expressions are important tools in natural language processing with a number of applications:
    - Tokenization
    - Stemming
    - Spell checking
    - Extracting information
    - Etc.

Using Regular Expressions in Python

- By practice, you can memorize
  - meta-characters
  - Wildcards
  - Ranges
  - Kleene Closures

Using Regular Expressions in Python

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*)

Using Regular Expressions in Python

Operator	Behavior
`?`	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

Exercise: Rotokas Analysis

- Rotokas is a language spoken by some 4,000 people (Wikipedia).
- It has the smallest alphabet in use (most probably!):
  - Only 10 lettes *A E G I K O P R S T U V*

Exercise: Rotokas Analysis

- Rotokas is a language spoken by some 4,000 people (Wikipedia).
- It has the smallest alphabet in use (most probably!):
  - Only 10 lettes *A E G I K O P R S T U V*
- We would like to extract all consonant-vowel sequences from the words of Rotokas, e.g. *ka*, si, ti, etc.
- A Rotokas dictionary is in NLTK distribution: `nltk.corpus.toolbox.words('rotokas.dic')`
- Use regular expressions to extract all the combination of consonant-vowels from this dictionary.

Exercise: Rotokas Analysis

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in
     re.findall(r'[ptksvr][aeiou]', w)]
```

Exercise: Rotokas Analysis

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in
     re.findall(r'[ptksvr][aeiou]', w)]
```

```
>>> for w in rotokas_words:
     for cv in re.findall(r'[ptksvr][aeiou]', w):
          cvs.append(cv)
```

Exercise: Rotokas Analysis

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in
     re.findall(r'[ptksvr][aeiou]', w)]
```

- Lets make a conditional frequency for the consonant and vowels:
  - This can be presented by a contingency table:
    - Each row represent a consonant and each column represent a vowel.
    - Each cell of table shows the count of occurrences of a vowel after a consonant.

	a	e	i	o	u
k					
p					
r					
s					
t					
v					

Exercise: Rotokas Analysis

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in
           re.findall(r'[ptksvr][aeiou]', w)]
>>> cfd = nltk.ConditionalFreqDist(cvs)
>>> cfd.tabulate()
```

	a	e	i	o	u
k	418	148	94	420	173
p	83	31	105	34	51
r	187	63	84	89	79
s	0	0	100	2	1
t	47	8	0	148	37
v	93	27	105	48	49

Exercise: Rotokas Analysis

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in
            re.findall(r'[ptksvr][aeiou]', w)]
>>> cfd = nltk.ConditionalFreqDist(cvs)
>>> cfd.tabulate()
```

	a	e	i	o	u
k	418	148	94	420	173
p	83	31	105	34	51
r	187	63	84	89	79
s	0	0	100	2	1
t	47	8	0	148	37
v	93	27	105	48	49

A number of tasks in statistical Natural Language Processing involves the study of this table.

Exercise: Rotokas Analysis

```
>>> rotokas_words = nltk.corpus.toolbox.words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words for cv in
          re.findall(r'[ptksvr][aeiou]', w)]
>>> cfd = nltk.ConditionalFreqDist(cvs)
>>> cfd.tabulate()
```

	a	e	i	o	u
k	418	148	94	420	173
p	83	31	105	34	51
r	187	63	84	89	79
s	0	0	100	2	1
t	47	8	0	148	37
v	93	27	105	48	49

In this example, by examining the rows for *s* and *t*, we see they are in partial “complementary distribution”

Exercise: Rotokas Analysis

```
>>> rotokas_words = nltk.corpus.toolbox words('rotokas.dic')
>>> cvs = [cv for w in rotokas_words f   r cv in
          re.findall(r'[ptks   ][aei       w)]
>>> cfd = nltk.Con      ionalF
>>> cfd.tabulate()
```

	a	e	i	o	u
		148	94	420	173
		31	105	34	51
	187	63	84	89	79
s		0	100	2	1
t	47	8	0	148	37
v	93	27	105	48	49

In this exam

the rows for s a

“compleme

T and *S* both represent the phoneme /t/, written with *S* before an *I* and in the name 'Rotokas', and with *T* elsewhere.

Exercise: Rotokas Analysis

```
>>> rotokas_words = nlt                              ds('rotokas.dic')
>>> cvs = [
```

			i	o	u
			94	420	173
			105	34	51
			84	89	79
		0	100	2	1
	47	8	0	148	37
v	93	27	105	48	49

Change the row and column of this table, to investigate, observer and justify various linguistic phenomenon, i.e. the research in statistical natural language processing

Searching Tokenized Text: RegEx over Tokens

- NLTK offers the unique functionality of defining regular expressions over lists of tokens:
  - The angle brackets <> are used to mark token boundaries.
  - RegEx patterns can be difnied over or within the <>
    - E.g. “<a> <man>” finds all instances of “a man” in text.
    - The pattern “<.*>” matches any single token

```
>>> import nltk
>>> from nltk.corpus import nps_chat
>>> chat = nltk.Text(nps_chat.words())
>>> chat.findall(r"<.*> <.*> <bro>")
you rule bro; telling you bro; u twizted bro
```

Searching Tokenized Text: RegEx over Tokens

- NLTK offers the unique functionality of defining regular expressions over lists of tokens:
  - The angle brackets <> are used to mark token boundaries.
  - RegEx patterns can be difnied over or within the <>
    - E.g. “<a> <man>” finds all instances of “a man” in text.
    - The pattern “<.*>” matches any single token

```
>>> import nltk
>>> from nltk.corpus import nps_chat
>>> chat = nltk.Text(nps_chat.words())
>>> chat.findall(r"<.*> <.*> <bro>")
you rule bro; telling you bro; u twizted bro
>>>
```

`nltk.Text` is a wrapper around a sequence of tokens

Searching Tokenized Text: RegEx over Tokens

- NLTK offers the unique functionality of defining regular expressions over lists of tokens:
  - The angle brackets <> are used to mark token boundaries.
  - RegEx patterns can be difnied over or within the <>
    - E.g. “<a> <man>” finds all instances of “a man” in text.
    - The pattern “<.*>” matches any single token

```
>>> import nltk
>>> from nltk.corpus import nps_chat
>>> chat = nltk.Text(nps_chat.words())
>>> chat.findall(r"<.*> <.*> <bro>")
you rule bro; telling you bro; u twizted bro
>>>
```

`findall` is then a function that enable us to search for RegEx patterns over tokens

Searching Tokenized Text: RegEx over Tokens

```
>>> chat.findall(r"<l.*>{3,}")
lol lol lol; lmao lol lol; lol lol lol; la la la la la;
la la la; la la la; lovely lol lol love; lol lol lol.;
la la la; la la la
```

Application Example for RegEx over Tokens

- Information Extraction
  - Searching a large text corpus for expressions of the form *x and other ys* allows us to discover hypernyms.
    - Hypernyms: superordinate, for example, *color* is a hypernym of *red.*

Application Example for RegEx over Tokens

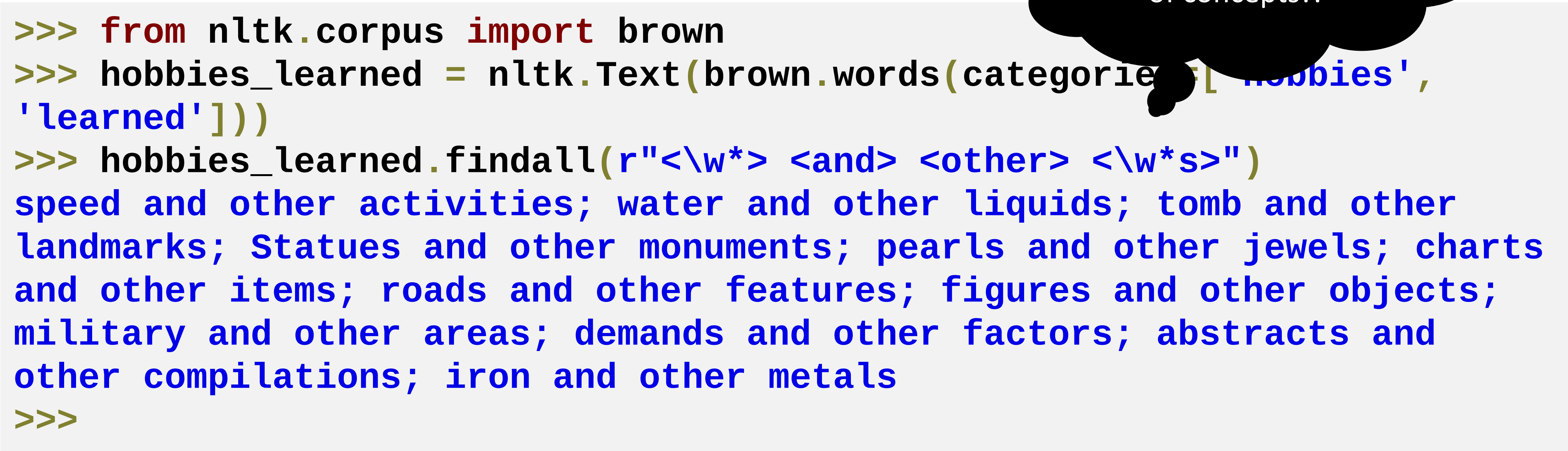
- Information Extraction
  - Searching a large text corpus for expressions of the form *x and other ys* allows us to discover hypernyms.
    - Hypernyms: superordinate, for example, *color* is a hypernym of *red.*

```
>>> from nltk.corpus import brown
>>> hobbies_learned = nltk.Text(brown.words(categories=['hobbies',
'learned']))
>>> hobbies_learned.findall(r"<\w*> <and> <other> <\w*s>")
speed and other activities; water and other liquids; tomb and other
landmarks; Statues and other monuments; pearls and other jewels; charts
and other items; roads and other features; figures and other objects;
military and other areas; demands and other factors; abstracts and
other compilations; iron and other metals
>>>
```

Application Example for RegEx over Tokens

- Information Extraction
  - Searching a large text corpus for expressions of the form x an ws us to discover hypernyms.
    - Hypernyms: superordinate, for example, *color* is a hyperny

Constructing an ontology/taxonomy of concepts!?



```
>>> from nltk.corpus import brown
>>> hobbies_learned = nltk.Text(brown.words(categorie =['hobbies',
'learned']))
>>> hobbies_learned.findall(r"<\w*> <and> <other> <\w*s>")
speed and other activities; water and other liquids; tomb and other
landmarks; Statues and other monuments; pearls and other jewels; charts
and other items; roads and other features; figures and other objects;
military and other areas; demands and other factors; abstracts and
other compilations; iron and other metals
>>>
```

Exercise

- Look for instances of the pattern *as x as y* to discover information about entities and their properties.

Exercise

- Look for instances of the pattern *as x as y* to discover information about entities and their properties.

```
r“<as> <\w*> <as> <\w*>"
```

Text Normalization

- Normalization is a process that aims to eliminate unwanted distinctions between text units:
  - Simple example of uppercase letters to lowercase letters: The, THE, the -> the
  - More sophisticate example of replacing names with the category of concept they represent: Passau, Munich, Galway, New York -> <CITY>

Text Normalization

- Normalization is a process that aims to eliminate unwanted distinctions between text units:
  - Simple example of uppercase letters to lowercase letters: The, THE, the -> the
  - More sophisticate example of replacing names with the category of concept they represent: Passau, Munich, Galway, New York -> <CITY>
- It is an application dependant process.
- Common text normalization:
  - Stemming
  - Lemmatization

Stemming & Lemmatization

- In many applications, e.g. search, word form is not important, e.g. laptop and laptops are both ok in search and information retrieval.
  - Both laptop and laptops are a from of the stem or lemma “laptop”.
  - Similarly, “walk, walking, walked” are **inflected** forms of lemma “walk”.
  - *Also, democracy*, *democratic*, and *democratization* are **derivationally** related.
- **Stemming** and **lemmatization** is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form.*

*http://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

Stemming & Lemmatization

- **Stemming** usually refers to a heuristic process that strips off the ends of words.
  - Usually it gives the correct answer, and often includes the removal of derivational affixes.
- **Lemmatization** is the more sophisticated way of getting word bases.
  - It uses a vocabulary and morphological analysis of words
  - It is aiming to remove inflectional endings only and to return the base or dictionary form of a word.

*http://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

Stemming Algorithms

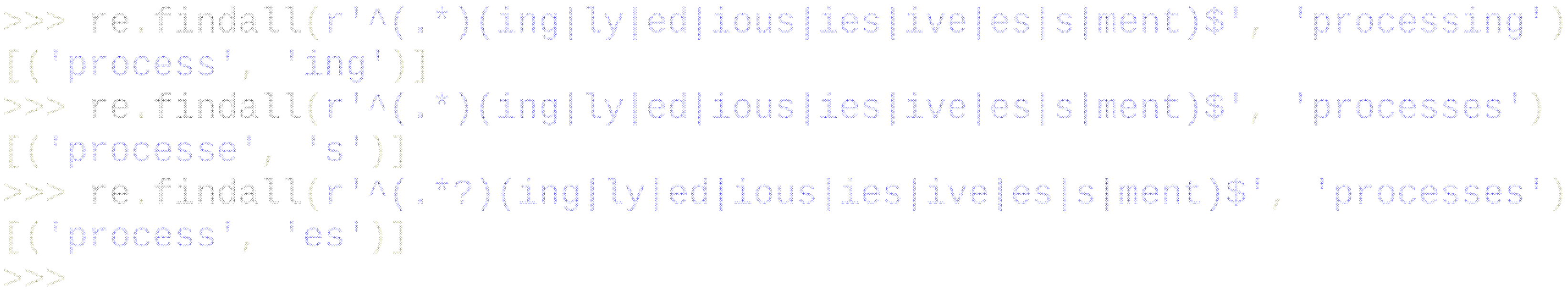
- A naïve approach for stemming is to simply strip off a set of suffixes

```
>>> def stemming(word):
     for suffix in ["ing", "ly", "ed", "ious", "ies", "ive",
"es", "s", "ment"]:
          if word.endswith(suffix): print word[:-len(suffix)]
>>> stemming("widely")
wide
>>> stemming("Lily")
Li
>>>
```

Stemming Algorithms

- We can also use regular expressions:

```
r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$'
```



```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
>>>
```

Stemming Algorithms

- We can also use regular expressions:

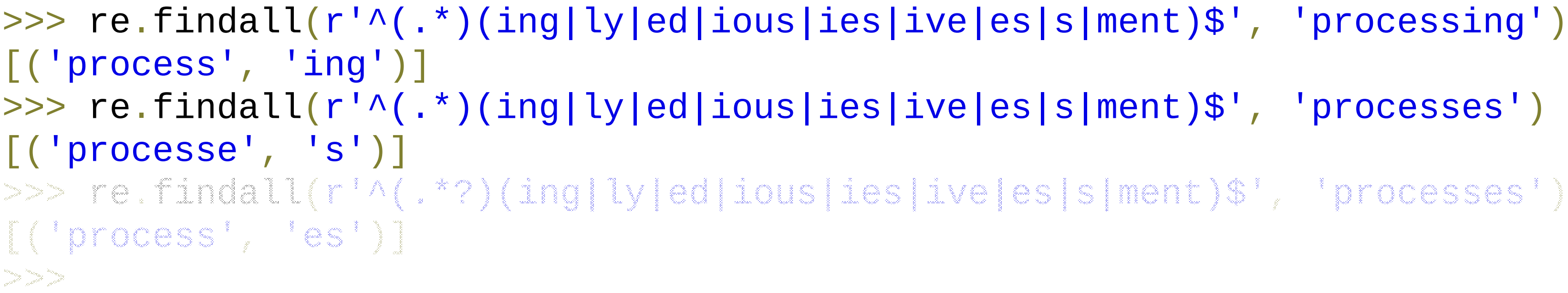
`r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$'`

```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
>>>
```

Stemming Algorithms

- We can also use regular expressions:

`r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$'`



```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
>>>
```

Stemming Algorithms

- We can also use regular expressions:

```
r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$'
```

```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
>>>
```

the star operator
is “greedy”

Stemming Algorithms

- We can also use regular expressions:

`r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$'`

```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
>>>
```

Stemming Algorithms

- We can also use regular expressions:

```
r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$'
```

```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
>>> re.findall(r'^(.*?)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('process', 'es')]
>>>
```

*? is the "non-greedy" operator

Stemming Algorithms

- We can also use regular expressions:

r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$'

```
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processing')
[('process', 'ing')]
>>> re.findall(r'^(.*)(ing|ly|ed|ious|ies|ive|es|s|ment)$', 'processes')
[('processe', 's')]
>>> re.findall(r'^(.*?)(ing|ly|ed|iou   es|i          en  $', 'processes')
[('process', 'es')]
>>>
```

Can you suggest inputs that result in false output?

Stemming Algorithms

- NLTK includes several stemmers.
- The *Porter* and *Lancaster* stemmers are rule-based algorithms for stripping affixes.

```
>>> porter = nltk.PorterStemmer()
>>> lancaster = nltk.LancasterStemmer()
>>> porter.stem("lying")
u'lie'
>>> lancaster.stem("lying")
'lying'
>>> porter.stem("arguing")
u'argu'
>>> lancaster.stem("arguing")
'argu'
>>>
```

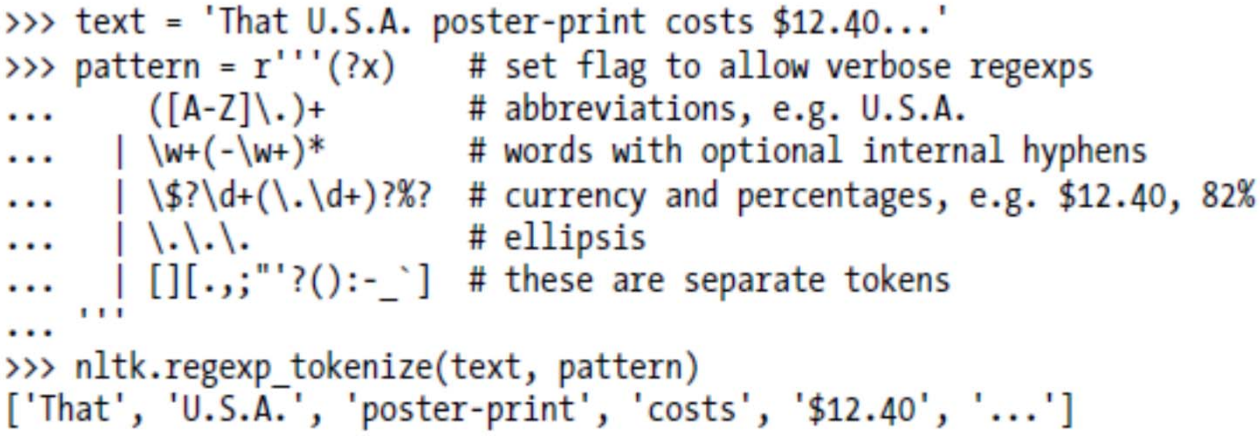
Lemmatization using NLTK

- The WordNet lemmatizer can also be used.
  - WordNet lemmatizer exploits WordNet dictionary (thus it is slower).

```
wnl = nltk.WordNetLemmatizer()
>>> wnl.lemmatize("arguing")
'arguing'
>>> wnl.lemmatize("arguing", pos=u'a')
'arguing'
>>> wnl.lemmatize("arguing", pos=u'v')
u'argue'
```

Text Tokenization

- Tokenization is a challenging task (discussed before)
- For a number of languages, regular expressions are handy tools to perform tokenization:



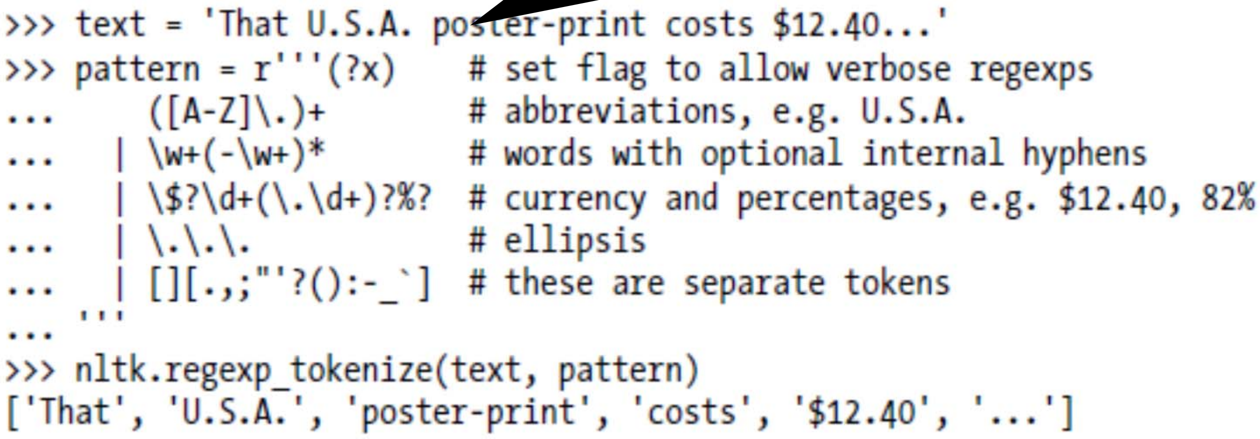
```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens
...  '''
...
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Text Tokenization

- Tokenization is a challeng
- For a number of languag
  perform tokenization:

Use the verbose mode to write more readable regular expressions:

- Whitespace within the pattern is ignored, use \s instead.
- All characters from the leftmost such '#' through the end of the line are ignored.



```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens
    '''
...
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Text Tokenization

- Tokenization is a challenging task (discussed before)
- For a number of languages, regular expressions are handy tools to perform tokenization:

Word Segmentation

- For some writing systems, tokenizing text is made more difficult by the fact that there is no visual representation of word boundaries.
  - We may mark word boundaries in other ways than list of tokens.
  - A research challenge on its own!

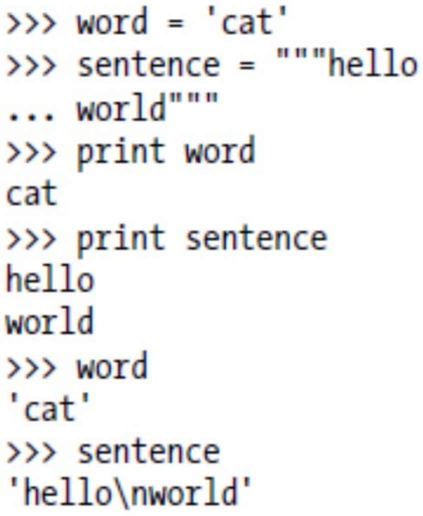
Generating Outputs: Lists to Strings

- The `join()` method can be used to convert lists to strings:

```
>>> silly = ['We', 'called', 'him', 'Tortoise', 'because', 'he', 'taught', 'us', '.']
>>> ' '.join(silly)
'We called him Tortoise because he taught us .'
>>> '*'.join(silly)
'We*called*him*Tortoise*because*he*taught*us*.'
>>>
```

Formatting Strings

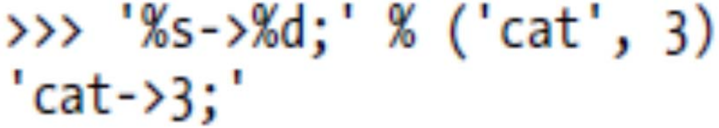
- The `print` command produce the most human-readable form of an object.
- Naming the variable at a prompt also shows us a string.



```
>>> word = 'cat'
>>> sentence = """hello
... world"""
>>> print word
cat
>>> print sentence
hello
world
>>> word
'cat'
>>> sentence
'hello\nworld'
```

Formatting Strings

- Formatted output, however, is often required to “export” our data:
    - Remember the dictionary exercise!
- “String formatting expressions**”** are used for print formatting:
    - The special symbols %s and %d are placeholders for strings and (decimal) integers.
    - The %s and %d symbols are called “conversion specifiers”**.**
    - The string containing conversion specifiers is called a “format string”.
    - The % operator and a tuple of values are combined to create a complete string formatting expression.

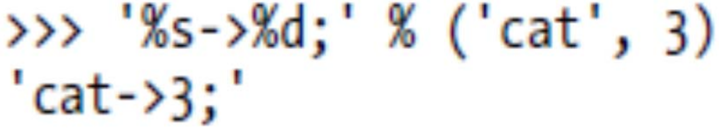


```
>>> '%s->%d;' % ('cat', 3)
'cat->3;'
```

Formatting Strings

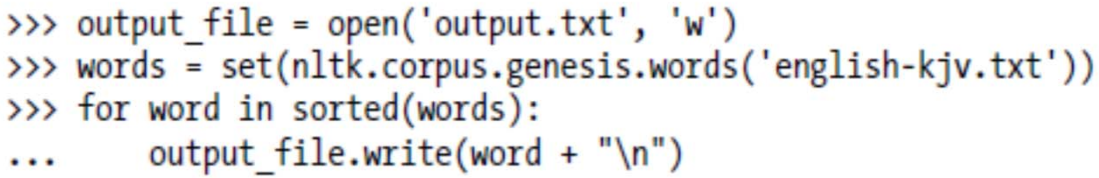
- Formatted output, however, is often required to “export” our data:
  - Remember the dictionary exercise!
- “String formatting expressions” are used for print formatting:
  - The special symbols %s and %d are placeholders for strings and (decimal) integers.
  - The %s and %d symbols are called “conversion specifiers”.
  - The string containing conversion specifier called ing”
  - The % operator and a tuple of valu string formatting expression.

Important application in tabulated data generation



```
>>> '%s->%d;' % ('cat', 3)
'cat->3;'
```

Writing Results to Files (reminder)

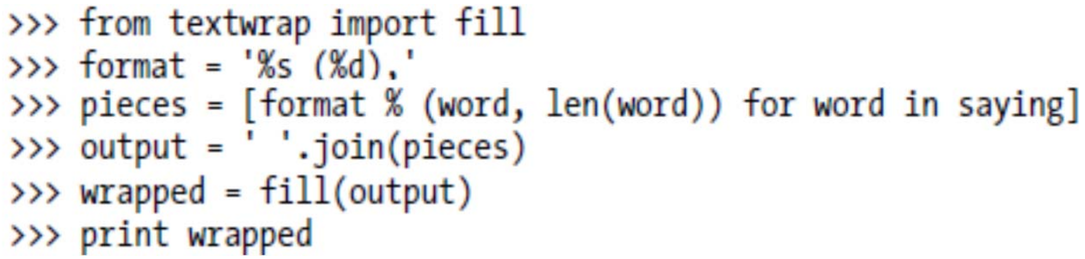


```
>>> output_file = open('output.txt', 'w')
>>> words = set(nltk.corpus.genesis.words('english-kjv.txt'))
>>> for word in sorted(words):
...     output_file.write(word + "\n")
```

- Remember to convert non-text data to text data before writing it into a file using `str()`.

Text Wrapping

- Python’s **`textwrap`** module can be used for wrapping lines:



```
>>> from textwrap import fill
>>> format = '%s (%d),'
>>> pieces = [format % (word, len(word)) for word in saying]
>>> output = ' '.join(pieces)
>>> wrapped = fill(output)
>>> print wrapped
```

Summary

- We know more about accessing raw text strings and processing it.
- We are able to use Regular Expressions for a number of tasks:
  - Tokenization
  - Stemming
  - At the token level for information extraction
- We now have an idea of stemming and lemmatization

Next Session

- For the next session, we cover chapter 4 of Natural Language Processing with Python to review basics such:
  - Designing algorithms
  - Structured Programming
  - Looking into a few Python libraries
- We will have a deeper look into NLTK's **`ConditionalFreqDist()`**
- We discuss proposed project titles
  - If you have your own title, please be ready to provide an overview, i.e.
    - What is the problem?
    - Why is it important?
    - What kind of result do you expect from you project work?