PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

FACULTAD DE LETRAS

Introducción a la lingüística computacional

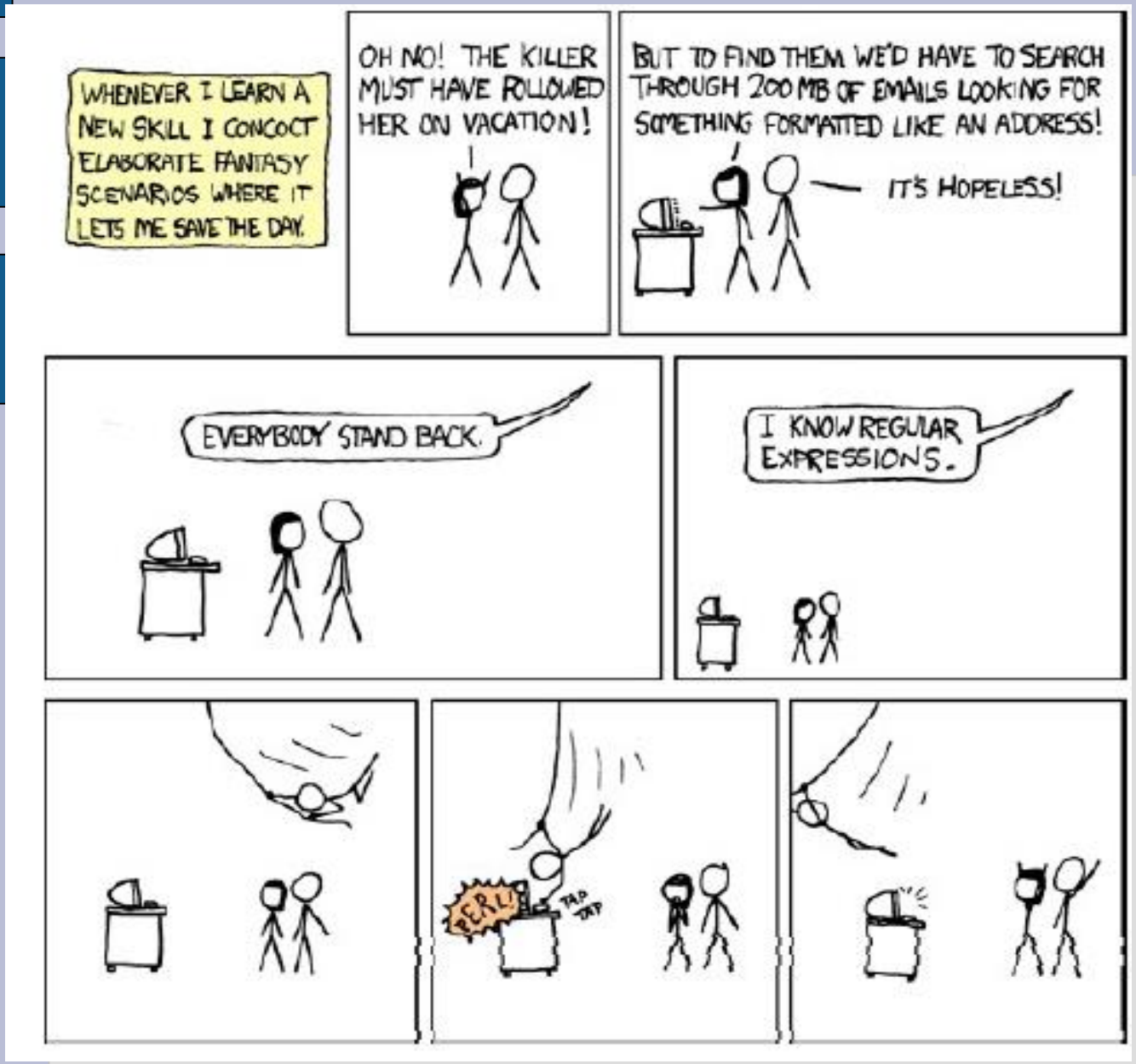
César Antonio Aguilar

Facultad de Lenguas y Letras

29/08/2017

Cesar.Aguilar72@gmail.com

Síntesis de la clase pasada (1)

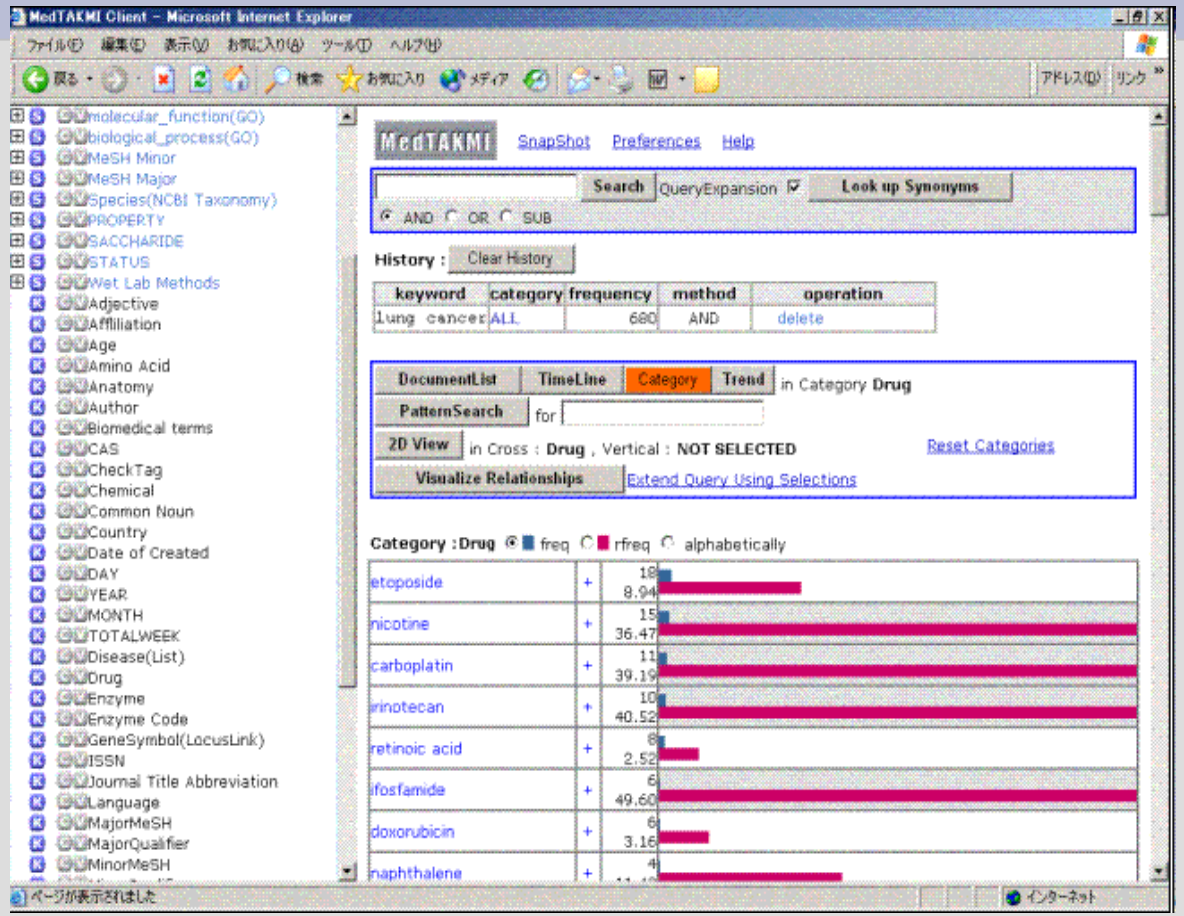


En la clase anterior empezamos a revisar, de manera general, el uso de expresiones regulares en Python.

Síntesis de la clase pasada (2)

Recordemos: una expresión regular es un tipo de patrón o estructura que describe un conjunto de cadenas sin enumerar sus elementos.

Es “regular porque su codificación tiende a ser repetitiva (de hecho, se pueden considerar una especie de “rutina” para comunicar cierta información).



Síntesis de la clase pasada (3)

¿Y para qué queremos usar expresiones regulares? Veamos el siguiente ejemplo: supongamos que queremos hacer búsquedas con el nombre *Händel*, y tenemos que lidiar con dos variantes de éste: *Handel* y *Haendel*.

Para resolver este problema, podríamos agrupar estas tres variantes bajo una estructura regular del tipo:

H(a|ä|ae)ndel

Esto equivale a decir algo: “una cadena formada por los caracteres 'H', las opciones 'a', o 'ä', o 'ae', 'n', 'd', 'e' y 'l'”. **Recuerden: no estamos trabajando con una palabra en sí, sino con una secuencia de signos.**

Ejercicio: buscando concordancias en un texto nuevo (1)

Volvamos a nuestra sesión anterior, y revisemos nuestro ejercicio para transformar una cadena de caracteres en una lista de palabras. Teníamos dos medios para hacer esto, los cuales nos generaban diferentes resultados:

```
In [22]: Lista_Ariel01 = re.split(r' ', Cadena_Ariel01)

In [23]: type(Lista_Ariel01)
Out[23]: list

In [24]: len(Lista_Ariel01)
Out[24]: 25004
```

```
In [25]: Lista_Ariel02 = re.split(r'\W+', Cadena_Ariel01)

In [26]: type(Lista_Ariel02)
Out[26]: list

In [27]: len(Lista_Ariel02)
Out[27]: 21606
```

Ejercicio: buscando concordancias en un texto nuevo (2)

Ahora, vamos a probar otros dos. El primero, implementando un listado de patrones empleado expresiones regulares. Para ello, primero analicemos el siguiente segmento:

```
[1]: import nltk, re

[2]: Text01 = 'That U.S.A. poster-print costs $12.40...'
```

Ejercicio: buscando concordancias en un texto nuevo (3)

Luego, añadimos nuestro listado de patrones:

```
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; includes ], [
…:  '''
```

Ahora escriban la siguiente instrucción:

```
nltk.regexp_tokenize(Text01, pattern)
```

Ejercicio: buscando concordancias en un texto nuevo (4)

Resultado:

```
In [4]: nltk.regexp_tokenize(Text01, pattern)
Out[4]: ['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

Nota: usando la instrucción anterior, lo que estamos haciendo es aplicar un criterio que nos permite reconocer **formas de palabras para el inglés**, las cuales vamos a identificar con el término **token** (esto es, se trata de una *muestra* derivada de una forma de palabra concreta).

Ejercicio: buscando concordancias en un texto nuevo (5)

Apliquemos esta instrucción y este listado de patrones al texto que analizamos en la sesión pasada:

```
Cadena_Ariel01 = open('C://Anaconda3/Corpus_Pruebas/ariel01.txt',
'rU').read()
```

Transformemos esta cadena en una lista de palabras, usando esta nueva instrucción:

```
Lista_Ariel03 = nltk.regexp_tokenize(Cadena_Ariel01, pattern)
```

Ejercicio: buscando concordancias en un texto nuevo (6)

Veamos los resultados:

```
In [7]: type(Lista_Ariel03)
Out[7]: list

In [8]: len(Lista_Ariel03)
Out[8]: 26421

In [9]: Rango_Ariel03 = Lista_Ariel03[0:50]

In [10]: print(Rango_Ariel03)
['THE', 'DEVELOPMENT', 'OF', 'PERSON', 'AGREEMENT', 'MARKERS', ':', 'FROM', 'PRONOUNS',
'TO', 'HIGHER', 'ACCESSIBILITY', 'MARKERS', 'Mira', 'Ariel', '(', '1999', ')', 'Tel-Aviv',
'University', '1', '.', 'From', 'free', 'pronouns', 'to', 'verbal', 'agreement', ':',
'Introduction', 'Grammatical', 'items', 'often', 'begin', 'their', 'linguistic', 'life',
'as', 'regular', 'lexical', 'items', '(', 'Meillet', '1912', ')', '.', 'Grammaticization',
'is', 'said', 'to']
```

Tokenización (1)

Finalmente, aprovechando esta técnica de identificación de formas de palabra, podemos sintetizar todos los métodos anteriores en uno, usando esta instrucción:

```
Tokens_Ariel01 = nltk.word_tokenize(Cadena_Ariel01)
```

¿Qué obtuvimos? Podemos deducirlo usando aplicando las siguientes funciones?

¿Qué tipo de objeto tenemos?

```
type(Tokens_Ariel01)
```

¿Cuántos objetos tenemos en nuestro conjunto?

```
len(Tokens_Ariel01)
```

Tokenización (2)

Veamos nuestros resultados:

```
In [11]: Tokens_Ariel01 = nltk.word_tokenize(Cadena_Ariel01)

In [12]: type(Tokens_Ariel01)
Out[12]: list

In [13]: len(Tokens_Ariel01)
Out[13]: 26348

In [14]: Rango_Ariel04 = Tokens_Ariel01[0:50]

In [15]: print(Rango_Ariel04)
['THE', 'DEVELOPMENT', 'OF', 'PERSON', 'AGREEMENT', 'MARKERS', ':', 'FROM', 'PRONOUNS',
'TO', 'HIGHER', 'ACCESSIBILITY', 'MARKERS*', 'Mira', 'Ariel', '(', '1999', ')', 'Tel-Aviv',
'University', '1', '.', 'From', 'free', 'pronouns', 'to', 'verbal', 'agreement', ':',
'Introduction', 'Grammatical', 'items', 'often', 'begin', 'their', 'linguistic', 'life',
'as', 'regular', 'lexical', 'items', '(', 'Meillet', '1912', ')', '.', 'Grammaticization',
'is', 'said', 'to']
```

Tokenización (3)

Ahora, una vez que hemos tokenizado nuestra cadena de caracteres (y que al final hemos convertido en una lista de tokens), podemos transformarla en un **texto** legible para el buscador de concordancias de NLTK, usando la siguiente instrucción:

```
In [16]: Text_Ariel01 = nltk.Text(Tokens_Ariel01)

In [17]: type(Text_Ariel01)
Out[17]: nltk.text.Text
```

Concordancias con nuevos textos (1)

Una vez que hemos generado nuestro texto, apliquemos algunas herramientas conocidas, p. e., ¿podemos generar una lista de posibles colocaciones? Veamos:

```
In [18]: Text_Ariel01.collocations()
third person; Accessibility Theory; typological markedness; verbal
forms; highly accessible; third persons; person referents;
first/second persons; University Press; John Benjamins; second person;
high accessibility; first/second person; full NPs; frequency-driven
morphologization; agreement markers; zero subjects; verbal agreement;
future tense; discourse topic
```

Concordancias con nuevos textos (2)

Ahora, busquemos algunas concordancias basadas en este listado de colocaciones. ¿Qué les parece la palabra *theory*?:

```
In [19]: Text_Ariel01.concordance("Theory")
Displaying 25 of 64 matches:
es this insight into his synchronic theory about pronoun uses , cites 18th and
achment ( 2 . 1 ) and accessibility theory ( 2 . 2 ) . I will suggest that bot
argument will be that Accessibility Theory can resolve both questions in a uni
d . I argue that only Accessibility Theory can account for this new developmen
 account , but not to Accessibility Theory . I also specify what would count a
. All we need add via Accessibility Theory is that this reduction is not merel
 problematic link for Accessibility Theory is the change from 8aiii to b . In
gether . According to Accessibility Theory , the motivation behind verbal pers
 their verb . 2 . 3 . Accessibility Theory versus NP detachment Logically spea
e differences between Accessibility Theory and NP detachment as explanations f
irst/second persons , Accessibility Theory provides the more plausible account
e plausible account . Accessibility Theory predicts that non-participants ( th
e better motivated by Accessibility Theory ( see section 5 ) . According to Ac
on 5 ) . According to Accessibility Theory , although the final stage of infle
ver , as predicted by Accessibility Theory , synthetic forms ( inflected ) are
flection , since only accessibility theory , which treats agreement markers as
ement primarily , and Accessibility Theory motivates ( overt ) first/second pe
arkers . According to Accessibility Theory , it is very high accessibility cas
 1992 ) Note that for Accessibility Theory , it is not really inter-person fre
s ) . As predicted by Accessibility Theory , it is full NPs that form the majo
ction is predicted by Accessibility Theory , since the speaker and the address
lience . According to Accessibility Theory , the referent in this case is pred
esented evidence that accessibility theory can indeed account for the prevalen
Thus , based on Levy ( 1995 ) , one theory predicts zero third person marking
futed by others . The Accessibility Theory account is the only one which is su
```

Concordancias con nuevos textos (3)

Veamos a continuación qué palabras se alternan con el término *accessibility*:

```
In [20]: Text_Ariel01.similar("accessibility")
frequency occurrence agreement pronouns person first grammaticization
inflection identity salience referents text degrees topic verbal
following zero persons number development
```

Usando la instrucción *common_contexts* obtenemos:

```
In [21]: Text_Ariel01.common_contexts(["accessibility", "salience"])
the_of of_of
```

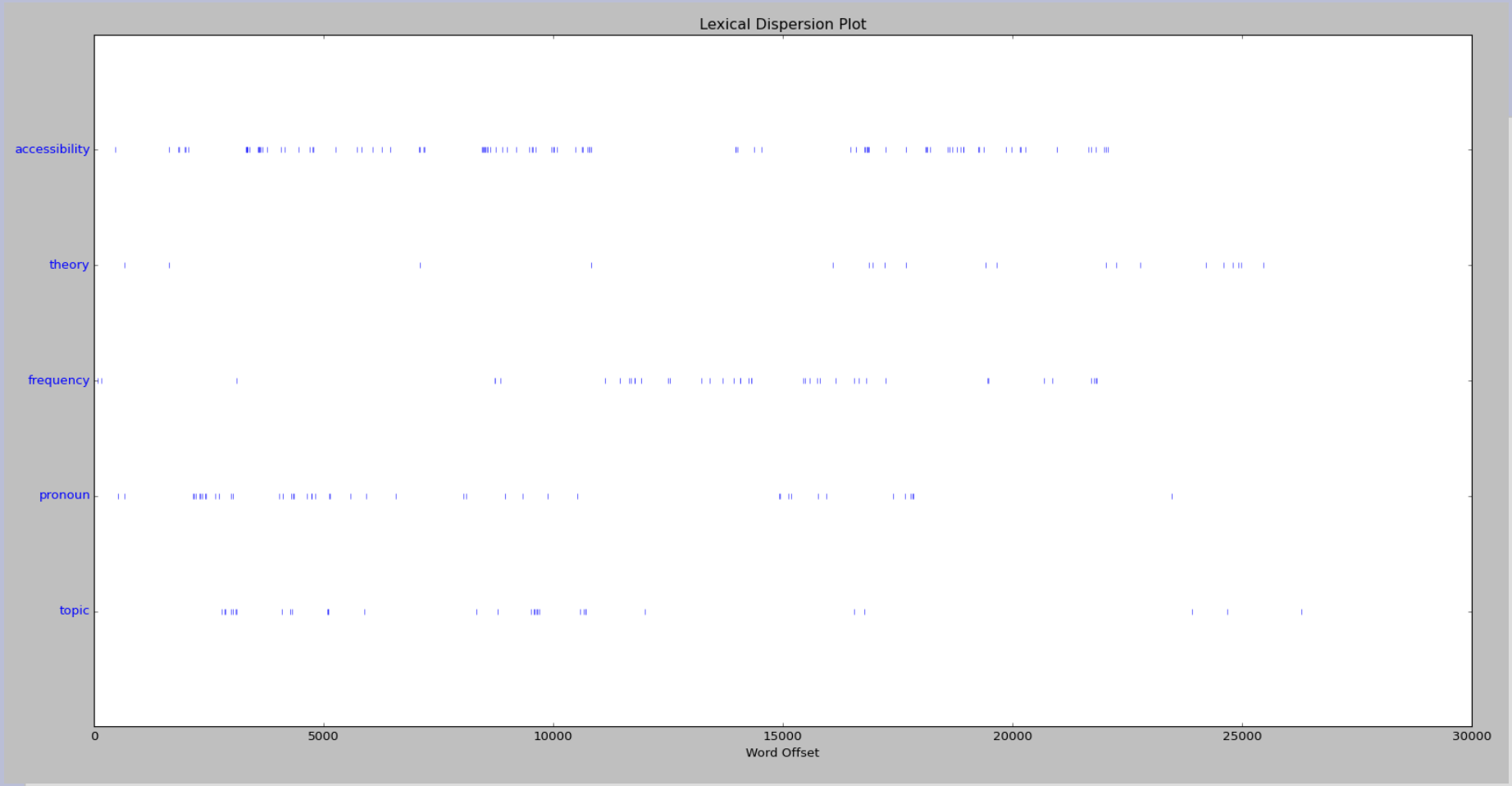
Concordancias con nuevos textos (4)

Finalmente, tratemos de generar un gráfico de dispersión. Consideremos estas palabras:

```
Text_Ariel01.dispersion_plot(["accessibility", "theory", "frequency", "pronoun", "topic"])
```

Veamos el resultado en la siguiente lámina:

Concordancias con nuevos textos (5)



Usando métodos con Regexp (1)

Para concluir, hagamos algunas operaciones empleando expresiones regulares. Veamos:

1. Escribamos la siguiente cadena: “Python es un lenguaje interpretado, orientado a objetos, para fines generales. Python resulta muy adecuado para manipular archivos de texto”.

2. Una vez que hayan creado su cadena, vamos a aplicar los siguientes procesos:

2.1. **capitalize()** : devuelve una copia de la cadena con el primer carácter en mayúscula. Escriban entonces: “Nombre_de_su_cadena”.capitalize()

Usando métodos con Regexp (2)

2.2. **count (sub[, start[, end]])** : devuelve cuántas veces aparece subcadena en la cadena principal o S [start:end]. Los argumentos opcionales *start* y *end* se interpretan según la notación de corte. Escriban: “Nombre_de_su_cadena”.count(‘de’).

2.3. **replace (old, new[, maxsplit])** : devuelve una copia de la cadena en la que se han sustituido todas las apariciones de *old* por *new*. Si se proporciona el argumento opcional *maxsplit*, sólo se sustituyen las primeras *maxsplit* apariciones. Escriban: “Nombre_de_su_cadena”.replace(‘de’,’del’)

2.4. **find (sub[, start[, end]])** : devuelve el menor índice de la cadena para el que *sub* se encuentre, de tal modo que *sub* quede contenido en el rango [start, end). Los argumentos opcionales *start* y *end* se interpretan según la notación de corte. Devuelve **-1** si no se halla *sub*. Escriban: “Nombre_de_su_cadena”.find(‘de’).

Usando métodos con Regexp (3)

2.5. **split ([sep [,maxsplit]])** : devuelve una lista de las palabras de la cadena, usando *sep* como delimitador de palabras. Si se indica *maxsplit*, se devolverán como mucho *maxsplit* valores (el último elemento contendrá el resto de la cadena). Si no se especifica sep o es None, cualquier espacio en blanco sirve de separador.

Escriban “Nombre_de_su_cadena”.split(), para generar un lista dividiendo la cadena en palabras (delimitadas por espacio).

Otro ejemplo: escriban “Nombre_de_su_cadena”.split(‘.’), para dividir la cadena en una lista de oraciones delimitadas por un punto.

Usando métodos con Regexp (4)

2.6. **endswith (suffix[, start[, end]])** : devuelve verdadero si la cadena finaliza con el sufijo *suffix* especificado, en caso contrario es falso. Si se da valor al parámetro opcional *start*, la comprobación empieza en esa posición. Si se da valor al parámetro opcional *end*, la comprobación finaliza en esa posición.

2.7. **startswith (prefix[, start[, end]])** : devuelve verdadero si la cadena comienza por prefix, en caso contrario, devuelve falso. Si se proporciona el parámetro opcional start, se comprueba la cadena que empieza en esa posición. Si se proporciona el parámetro opcional end, se comprueba la cadena hasta esa posición.

Usando métodos con Regexp (5)

3. Ahora, para terminar, vamos a implementar dos métodos más o menos complejos para hacer dos tareas: encontrar y enlistar objetos (o *findall*) y substracción de objetos (o *sub*) dentro de una cadena, esto es:

3.1. Tarea01 = **re.findall(r'\b[p|P][a-z]*',** Cadena01**)**

3.2. Tarea02 = **re.sub(r'(\b[a-z]+) \1', r'\1'**, Cadena01)

¿Qué nos arroja como resultado? Veamos:

Usando métodos con Regexp (6)

```
In [28]: Tarea01 = re.findall(r'\b[p|P][a-z]*', Cadena01)

In [29]: print(Tarea01)
['Python', 'para', 'Python', 'para']

In [30]: Tarea02 = re.sub(r'(\b[a-z]+) \1', r'\1', Cadena01)

In [31]: print(Tarea02)
Python es un lenguaje interpretado, orientado a objetos, para fines generales. Python resulta muy
adecuado para manipular archivos de texto
```

Gracias por su atención

Blog del curso:
http://cesaraguilar.weebly.com/curso-de-procesamiento-del-lenguaje-natural.html