

PLN con Python

Alejandro Pimentel

Introducción

Objetivos del PLN

- ▶ Crear aplicaciones que puedan manipular, interpretar y generar lenguaje humano
- ▶ Modelar la capacidad lingüística humana.
- ▶ Representar el conocimiento lingüístico de una manera computacional.

- ▶ Introducción al lenguaje de programación Python.
- ▶ Instrucción a la programación y módulos con un enfoque para PLN.
 - ▶ Expresiones regulares
 - ▶ NLTK (natural language tool kit).

- ▶ Intérprete Python
- ▶ Editor de texto
- ▶ Internet



ANACONDA®

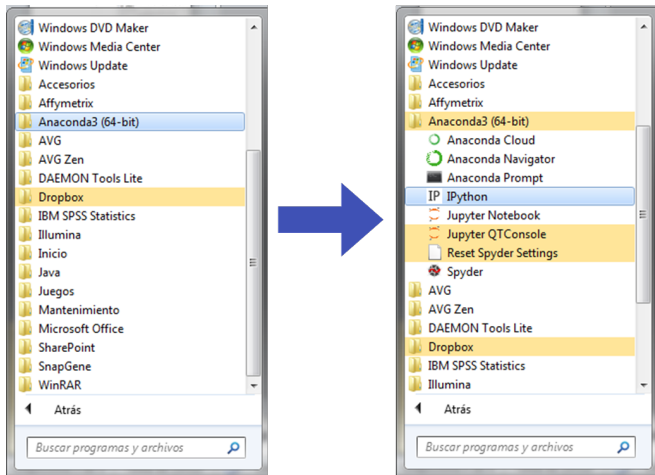


spyder

<https://www.continuum.io/downloads>

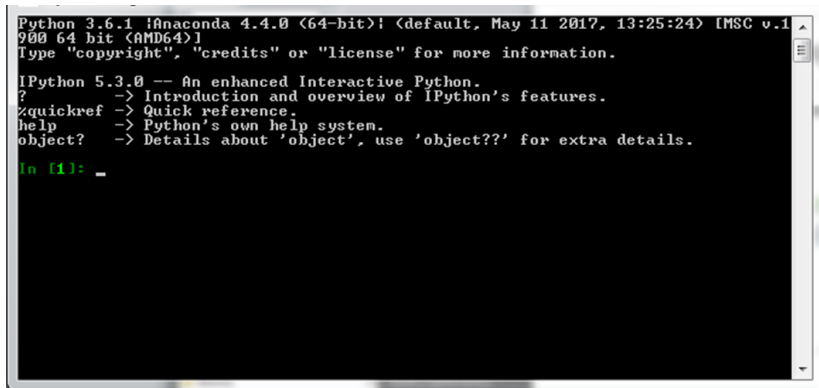
Primero lo primero

Depende de la forma en la que hayan instalado python, pueden comenzar a usarlo de diferentes maneras, como por la línea de comandos (cmd). Si instalaron anaconda, podrán entrar desde allí.



Python en consola

- ▶ Esta pantalla es la consola de python.



```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license()" for more information.

IPython 5.3.0 -- An enhanced Interactive Python.
?                -> Introduction and overview of IPython's features.
%quickref        -> Quick reference.
help             -> Python's own help system.
object?         -> Details about 'object', use 'object??' for extra details.

In [1]: _
```

- ▶ Aquí podrán escribir los comandos directamente para que python los interprete y ejecute.

Hola Mundo

en Python

- ▶ El primer programa por excelencia, el "Hola Mundo"
- ▶ La consola recibe la instrucción de mostrar en la pantalla un mensaje.

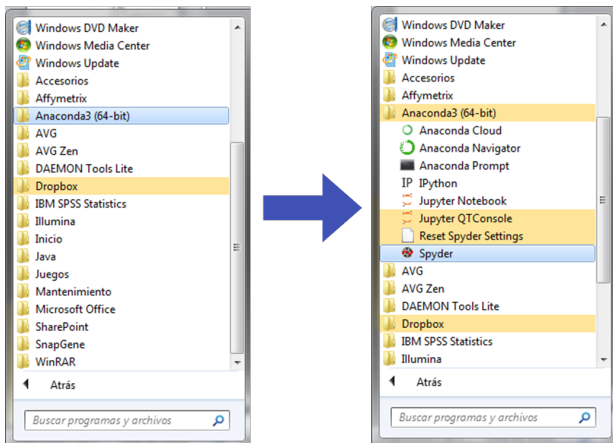
```
print("Hola Mundo")
```

- ▶ El mensaje debe ir entre comillas.

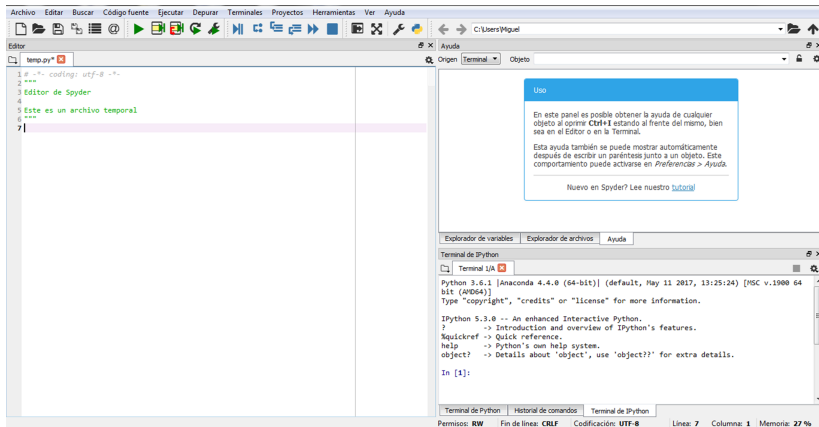
El editor

Python, al igual que la mayoría de los lenguajes de programación, no necesita nada más complejo que un bloc de notas. Pero existen un sinnúmero de opciones, pueden usar la que más les guste.

Si instalaron anaconda, y no tienen un editor preferido, pueden probar *Spyder*.



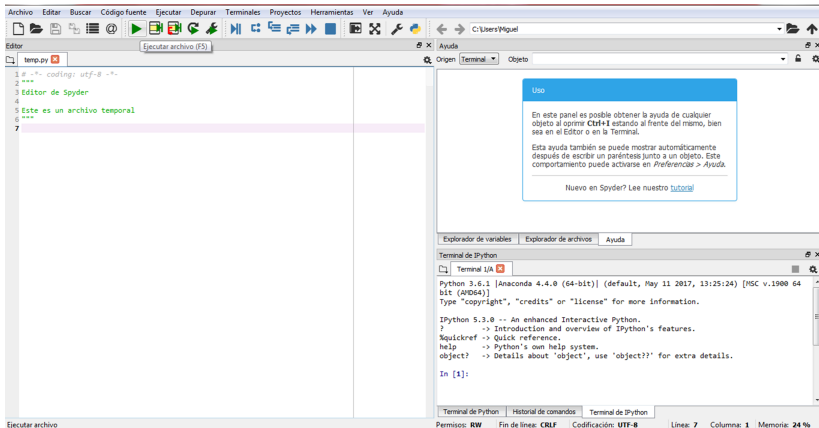
- ▶ Spyder cuenta con una consola en la que se pueden ejecutar comandos, o lo que se esté escribiendo en el editor.
- ▶ Además, cuenta con un apartado de ayuda para las funciones de Python.



- ▶ Dentro del editor podemos escribir una serie de comandos que queremos que se ejecuten.
- ▶ Serán leídos e interpretados en orden, uno tras otro.

```
print("De ola en ola,")  
print("de rama en rama,")  
print("el viento silba")  
print("cada mañana.")
```

- ▶ Una vez que se escriben las instrucciones, se puede ejecutar el archivo completo.
- ▶ Varios editores cuentan con esta función.



Operaciones

- ▶ Python también es capaz de hacer operaciones matemáticas, cual si fuera una calculadora simple.

```
print(2 + 2)
print(3 * 4)
print(100 - 1)
```

- ▶ Es importante notar que en este caso, no se usan comillas en el interior de la función `print()`.

- La razón, es que las comillas se utilizan cuando se está manejando texto, los números, para hacer operaciones, se deben tomar como números.

```
print("2" + "2")  
print("3 * 4")  
print("100" - 1)
```

Operaciones

- ▶ Ambas formas se pueden combinar dentro de la función `print()`.

```
print("2 + 2 =", 2 + 2)
print("3 * 4 =", 3 * 4)
print("100 - 1 =", 100 - 1)
```

- ▶ Es importante notar que el texto y los números están separados por una coma.

Entonces se tienen:

- ▶ Palabras, el texto que se debe manejar entre comillas.
- ▶ Números y operaciones, que deben ir sin comillas para que se evalúen.

¿Qué hay de las palabras sin comillas?

```
print(Hola)
```

Variables

- ▶ Las variables son palabras (o cualquier combinación de letras en realidad) que se utilizan para guardar valores.
- ▶ Cualquier valor o el resultado de cualquier operación puede ser asignado a una variable.
- ▶ Los valores se asignan con el símbolo =

```
nombre = "Juan Hernández Hernández"  
suma = 2 + 3  
print("Hola" , nombre)  
print("2 + 3 =", suma)
```

Variables

- ▶ Las variables también pueden ser utilizadas para asignar otras variables.
- ▶ O incluso para cambiar su propio valor

```
x = 2 + 3
y = x + 5
print("x =", x)
print("y =", y)

y = y + 10
print("y =", y)
```

IMPORTANTE

Dejen que su código les hable.

Y háganlo aunque les de flojera.

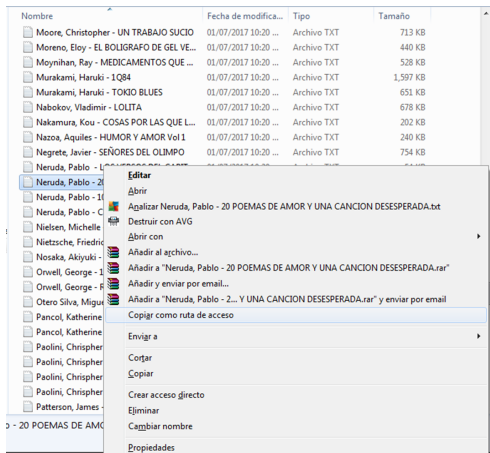
```
x = 2 + 3 # A x se le asigna el resultado de una suma.  
y = x + 5 # Se puede usar una variable asignada para hacer  
          # operaciones.  
print("x =", x) # El comando print() sirve para  
                # mostrar cosas en pantalla.  
print("x + 5 =", y) # Puede recibir texto o números,  
                   # incluso operaciones, pero separados por  
                   # comas.
```

Archivos

- ▶ Un corpus es un conjunto de documentos (texto, video, audio, etc.) destinado a la investigación.
- ▶ La recolección de un corpus depende de lo que se busca estudiar.
- ▶ Para este curso, contamos con un conjunto de documentos legales, sin embargo, si cuentan con un corpus propio pueden utilizarlo.

Ruta de archivos

- ▶ Es común necesitar la ruta de los archivos.
- ▶ En Windows, se puede usar **SHIFT+CLICK_DERECHO** en un archivo, lo que despliega un menú de opciones en el que aparece: **"Copiar como ruta de acceso"**



- Para comenzar a trabajar con un archivo, lo primero que se tiene que hacer es abrirlo.

IMPORTANTE Las rutas de los archivos en Windows separan las carpetas con diagonal invertida ('\\'). Éste es un símbolo especial que no se puede usar así nada mas, más adelante veremos por qué. Por ahora solo recuerden que deben duplicar la diagonal invertida ('\\' en lugar de '\\').

```
archivo_abierto=open("C:\\\\RUTA\\P_IPT_290216_73_Acc.txt")
```


- ▶ Ya que el archivo está abierto podemos leer el contenido.

```
texto=archivo_abierto.read()
```

- ▶ `read()` es una función de los archivos (debe ser un archivo que haya sido abierto con la función `open()`) que lee el texto y lo "regresa". Esto quiere decir que el resultado puede (o mejor dicho **debe**) asignarse a una variable, en este caso, la variable `texto`
- ▶ Es una función que no recibe ningún argumento, por eso los paréntesis vacíos.

- ▶ Podemos hacer que Python nos muestre el contenido del archivo, que ahora está en la variable `texto`.

```
print(texto)
```

- ▶ Recuerden que en esta ocasión NO se usan comillas, queremos usar la variable `texto` no el texto: *"texto"*
- ▶ Finalmente, es importante cerrar el archivo que abrimos en primer lugar:

```
archivo_abierto.close()
```

- ▶ El programa completo se vería algo así:

```
archivo_abierto=open("C:\\\\RUTA\\\\P_IFT_290216_73_Acc.txt" )  
texto=archivo_abierto.read()  
print(texto)  
archivo_abierto.close()
```

- ▶ Más sus respectivos comentarios

Escritura de archivos

- ▶ El proceso de escritura es similar, pero se distingue desde la forma en la que se abre el archivo

```
archivo_abierto=open("C:\\\\RUTA\\mi_archivo_nuevo.txt","w")
```

- ▶ Noten dos cosas.
- ▶ Primero, la 'w' como segundo argumento de la función `open()`. Esa 'w' significa *write*, y se usa para abrir el archivo en modo escritura (si se omite, como en el caso anterior, por defecto los archivos se abren en modo lectura: `r`).
- ▶ Segundo, el nombre del archivo. NO hagan esto con un archivo existente. Cuando se abre un archivo en modo escritura, se crea como nuevo, y Python no pregunta si quieres sobrescribir, lo hace.

Escritura de archivos

- ▶ Para escribir en el archivo, usaremos la función `write()`.

```
archivo_abierto=open("C:\\\\RUTA\\mi_archivo_nuevo.txt","w")
archivo_abierto.write("Esto se escribe en el archivo")
archivo_abierto.write("Esto tambien")
```

- ▶ La función `write()` es muy parecida a `print()`, con la diferencia de que manda el texto al archivo en lugar de a la pantalla.
- ▶ Para ver los cambios en su archivo, no olviden cerrarlo.

```
archivo_abierto.close()
```

Escritura de archivos

- ▶ Si abren su archivo verán lo que escribieron, y lo más probable es que se topen con algo como esto.

```
Esto se escribe en el archivoEsto tambien
```

- ▶ Probablemente este no era el resultado que esperaban.
- ▶ Una diferencia entre `print()` y `write()` es que `write()` no añade un salto de línea en el archivo como `print()` en la pantalla.

Escritura de archivos

- ▶ Para agregar saltos de línea a voluntad se usa el símbolo: `\n`
- ▶ Por lo tanto, el programa anterior, sería mejor escribirlo algo así:

```
archivo_abierto=open("C:\\\\RUTA\\mi_archivo_nuevo.txt","w")

archivo_abierto.write("Esto se escribe en el archivo\n")
archivo_abierto.write("Esto tambien\n")
archivo_abierto.write("Mira, puedo escribir \"comillas\"\\n")
archivo_abierto.write("Gracias a la diagonal invertida: \\ \n")

archivo_abierto.close()
```

- ▶ Al agregar un `\n` al final de cada texto en el que se use la función `write()`, se obtiene el mismo comportamiento que con `print()`
- ▶ Como ven, la diagonal invertida es un símbolo especial. También se puede usar para escribir comillas dentro de los mensajes y que no se malinterpreten como el cierre de comillas.

Ejemplo

```
c="C:\\Users\\user\\Desktop\\Documentos\\"
e="P_IFT_290216_73_Acc.txt"
s="archivo_nuevo.txt"
e2=open(c+e,"r")
s2=open(c+s,"w")
t=e2.read()
t2=t
s2.write(t2)
e2.close()
s2.close()
```


Ejemplo

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"
salida_nombre="archivo_nuevo.txt"

entrada_abierto=open(carpeta_nombre+archivo_nombre,"r")
salida_abierto=open(carpeta_nombre+salida_nombre,"w")

texto_entrada=entrada_abierto.read()
texto_salida=texto_entrada

salida_abierto.write(texto_salida)

entrada_abierto.close()
salida_abierto.close()
```

Manejo de archivos

- ▶ Aquí presentaré un código con otra manera de trabajar con un archivo abierto.
- ▶ Esta forma tiene la ventaja de ser mas clara.
- ▶ Y también más fácil ya que maneja por si misma el cierre del archivo.

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()
    print(texto)

# El programa continúa acá...
```

Manejo de archivos

- ▶ Aquí presentaré un código con otra manera de trabajar con un archivo abierto.
- ▶ Esta forma tiene la ventaja de ser mas clara.
- ▶ Y también más fácil ya que maneja por si misma el cierre del archivo.

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()
    print(texto)

# El programa continúa acá...
```

Programación

El 'if'

- ▶ El `if` es una instrucción que ejecuta un bloque si una condición se cumple.
- ▶ Es la instrucción más simple de control

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_200416_162_Acc.txt"
palabra="acuerdo"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

if palabra in texto:
    print("Encontré la palabra!")
if 2 > 5 :
    print("Dos es mayor que cinco!?")
```

if ... else

- ▶ Ya que estamos viendo cómo funciona el `if` veamos también una instrucción que trabaja en conjunto, el `else`

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_200416_162_Acc.txt"
palabra="ACUERDO"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

if palabra in texto:
    print("Encontré la palabra!!")
else:
    print("No encontré la palabra :(")
```

- ▶ Si la condición se cumple, se ejecuta el bloque del `if` y no el del `else`. Si la condición no se cumple, no se ejecuta el del `if` pero sí el del `else`

if ... elif ... else

- Y para completar, la instrucción: **elif**

```
palabra1="ACUERDO"
palabra2="RESOLUCIÓN"

with open(carpeteta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

if palabra1 in texto:
    print("Encontré la palabra 1!!")
elif palabra2 in texto:
    print("No está la palabra 1 pero si la 2")
else:
    print("No hay ninguna de tus palabras :(")
```

- **elif** es la combinación de un **else** con un **if**, se usa como **else**, y es ignorado si el primer **if** se cumple, pero a su vez, también revisa la verdad de una condición para ejecutar su bloque de código.

- ▶ Modifiquen el ejemplo anterior para que el programa les indique si el documento es un acuerdo, una resolución o si es otra cosa.
 - ▶ Utilicen la variable. Es decir, si intercambian el contenido de `palabra1` y `palabra2` su programa debe seguir indicando correctamente qué tipo de documento es.

Listas

- ▶ Algo interesante, las variables pueden tener más de un valor.
- ▶ Una variable, puede ser una lista.

```
semana_laboral=["Lunes","Martes","Miércoles","Jueves","Viernes"]  
print("Semana laboral =",semana_laboral)  
print("Dia 1 =", semana_laboral[0])  
print("Dia 2 =",semana_laboral[1])  
print("Dia 3 =",semana_laboral[2])  
print("Dia 4 =",semana_laboral[3])  
print("Dia 5 =",semana_laboral[4])
```

- ▶ Las listas pueden agrupar un conjunto de datos relacionados entre sí.
- ▶ **IMPORTANTE:** los índices de las listas, comienzan en 0.

- Si utilizamos los índices de la lista, los valores que contiene se pueden manipular individualmente como cualquier otra variable.

```
semana_laboral[4]="Sabado?"  
print("Semana laboral cambiada =",semana_laboral)
```

Listas

- Las listas tienen una serie de comandos propios que es muy útil tener en cuenta:

```
longitud_lista=len(semana_laboral)
print("Tamaño de la lista =",longitud_lista)

num_elemento=semana_laboral.index("Martes")
print("Lugar del Martes =",num_elemento)

semana_laboral.append("Viernes")
print("Semana laboral (append(\"Viernes\")) =",semana_laboral)

del semana_laboral[4]
print("Semana laboral (sin el lugar 4)) =",semana_laboral)

semana_laboral.remove("Viernes")
print("Semana laboral (sin \"Viernes\") =",semana_laboral)
```

Listas

- Las listas tienen una serie de comandos propios que es muy útil tener en cuenta:

```
longitud_lista=len(semana_laboral)
print("Tamaño de la lista =",longitud_lista)

num_elemento=semana_laboral.index("Martes")
print("Lugar del Martes =",num_elemento)

semana_laboral.append("Viernes")
print("Semana laboral (append(\"Viernes\")) =",semana_laboral)

del semana_laboral[4]
print("Semana laboral (sin el lugar 4)) =",semana_laboral)

semana_laboral.remove("Viernes")
print("Semana laboral (sin \"Viernes\") =",semana_laboral)
```

Listas

- ▶ Y ahora que ya sabemos de listas, hay que utilizarlas.
- ▶ Una forma muy intuitiva de manejar archivos, es por líneas. Python sabe, por supuesto, leer un archivo por líneas y ponerlas en una lista:

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="DOF_P_IPT_290216_71_Datos_Relevantes_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    lineas_lista=archivo.readlines()

print(lineas_lista)
```

- ▶ Muchos documentos tienen cierta estructura, e información separada en líneas, como las listas o las tablas.

Bucles

'for'

- ▶ Los bucles son los procesos iterativos.
- ▶ La misma acción se hace una y otra vez para un conjunto de datos, por lo general para todos los elementos de una lista.
- ▶ Por ejemplo, podemos hacer que para cada línea de un archivo Python la escriba en la pantalla:

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\Legales\\"
archivo_nombre="DOF_P_IFT_290216_71_Datos_Relevantes_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    lineas_lista=archivo.readlines()

for linea in lineas_lista:
    print(linea)
    print()
```

Bucles

'for'

- Veamos un ejemplo un poco mas complejo:

```
01: carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
02: archivo_nombre="DOF_P_IPT_291116_672_Acc.txt"
03:
04: with open(carpeta_nombre+archivo_nombre,"r") as archivo:
05:     lineas_lista=archivo.readlines()
06:
07: num_linea=1
08: for linea in lineas_lista:
09:     print("LINEA",num_linea,":",linea)
10:     num_linea=num_linea+1
11:
12: print("FIN DE ARCHIVO")
```

Bucles

'for'

- Veamos un ejemplo un poco mas complejo:

```
01: carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
02: archivo_nombre="DOF_P_IPT_291116_672_Acc.txt"
03:
04: with open(carpeta_nombre+archivo_nombre,"r") as archivo:
05:     lineas_lista=archivo.readlines()
06:
07: num_linea=1
08: for linea in lineas_lista:
09:     print("LINEA",num_linea,":",linea)
10:     num_linea=num_linea+1
11:
12: print("FIN DE ARCHIVO")
```


Bucles

'for'

- Veamos un ejemplo un poco mas complejo:

```
01: carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
02: archivo_nombre="DOF_P_IPT_291116_672_Acc.txt"
03:
04: with open(carpeta_nombre+archivo_nombre,"r") as archivo:
05:     lineas_lista=archivo.readlines()
06:
07: num_linea=1
08: for linea in lineas_lista:
09:     print("LINEA",num_linea,":",linea)
10:     num_linea=num_linea+1
11:
12: print("FIN DE ARCHIVO")
```

Bucles

'for'

- Los bloques de código se pueden anidar.

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="DOF_P_IPT_291116_672_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    lineas_lista=archivo.readlines()

num_linea=1
for linea in lineas_lista:
    if linea == "":
        pass
    else:
        print("LINEA",num_linea,":",linea)
        num_linea=num_linea+1

print("FIN DE ARCHIVO")
```

Bucles

'for'

- El código anterior "falla" para eliminar las líneas vacías, probemos este otro:

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="DOF_P_IPT_291116_672_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    lineas_lista=archivo.readlines()

num_linea=1
for linea in lineas_lista:
    if linea.strip() == "":
        continue
    print("LINEA",num_linea,":",linea)
    num_linea=num_linea+1

print("FIN DE ARCHIVO")
```

Ejercicio

Por orden de dificultad, modifiquen el programa para que:

- ▶ Cuente las líneas que contiene el archivo y lo indique.
- ▶ Cuando encuentre una línea que NO contenga texto, indique que esa línea esta vacía.
- ▶ Muestre cuántas líneas tienen texto y cuántas no.

Lista de archivos

No solo las oraciones se pueden manejar en listas:

```
import os

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"

archivos_lista=os.listdir(carpeta_nombre)

for archivo_nombre in archivos_lista:
    print(archivo_nombre)
```

- ▶ Aquí, hacemos uso de una nueva instrucción: el **import**
- ▶ Python tiene muchísimos complementos que tienen diferentes funciones, no los tiene precargados todos porque sería algo demasiado pesado. Es por eso que se utiliza el **import** cuando se quieren utilizar funciones que no están en el "paquete básico" de Python.

Lista de archivos

No solo las oraciones se pueden manejar en listas:

```
import os

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"

archivos_lista=os.listdir(carpeta_nombre)

for archivo_nombre in archivos_lista:
    print(archivo_nombre)
```

- ▶ En este caso, importamos el módulo `os`, que contiene funciones para interactuar con el sistema operativo.
- ▶ La función `listdir()` del módulo `os` permite obtener el contenido de una carpeta en forma de lista.

Lista de archivos

- Ahora podemos ejecutar comandos para todos los archivos de nuestra carpeta de trabajo.

```
import os

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivos_lista=os.listdir(carpeta_nombre)

for archivo_nombre in archivos_lista:
    archivo=open(carpeta_nombre+archivo_nombre)
    lineas_lista = archivo.readlines()
    archivo.close()
    longitud = len(lineas_lista)
    print("El archivo",archivo_nombre,"tiene",longitud,"lineas")
```

Es común que se quieran analizar documentos relacionados como si fueran uno solo, ya sea que sean documentos del mismo autor, mismo tema, o alguna característica común.

Nosotros tenemos documentos legales del Instituto Federal de Telecomunicaciones, el ejercicio consiste en:

- ▶ Obtener un documento (por ejemplo `"UNION.txt"`) que contenga la unión de todos los documentos del corpus.
 - ▶ Recuerden que pueden hacer iteraciones sobre sus documentos.
 - ▶ Utilicen una variable para ir acumulando el texto de los documentos.
 - ▶ Recuerden que pueden concatenar texto con el operador `'+'`

Segmentacion

Lista de palabras

- ▶ Ya que sabemos que podemos trabajar con nuestros archivos uno tras otro, volvamos a trabajar enfocándonos en uno.
- ▶ Esta vez, separaremos palabras.

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

palabras_lista=texto.split()
print(palabras_lista)
```

Lista de palabras

- Por cierto, podemos ordenar una lista, con su función `sort()`.

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

palabras_lista=texto.split()
palabras_lista.sort()
for palabra in palabras_lista:
    print(palabra)
```

Lista de palabras

- ▶ La función `split()` lo que hace es dividir el texto según los espacios que encuentra.
- ▶ Esta es la forma mas simple de separar palabras.
- ▶ Pero seguramente notarán algunos detalles del proceso. Principalmente, notarán que muchas palabras de la lista también tienen símbolos de puntuación como comas, puntos y paréntesis que no forman parte de la palabra. Es necesario un método más detallado y confiable para la separación.
- ▶ En PLN, a este proceso de segmentar el texto se llama *tokenización*. Y a los elementos obtenidos (palabras, signos de puntuación, urls, siglas, fechas, etc.) se le llaman *tokens*.

¿Cómo hacer un tokenizador?

- ▶ Tenemos que ser capaces de separar las palabras de la puntuación con la que colindan.
- ▶ Sin embargo, hay que considerar que no toda la puntuación se debe separar, tal es el caso de los números, las fechas, las siglas, entre otros.

Tokenización

forma básica

- ▶ De nuestra lista de palabras, podemos observar que en la mayoría de los casos, lo único que obstruye a las palabras son símbolos como paréntesis, comas, puntos y punto y comas.
- ▶ Un tokenizador sumamente básico se lograría remplazando esos signos para separarlos del texto, y más adelante usar la función `split()`.

Tokenización

forma básica

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

simbolos=["(",")",".",",",";",":","\\",""]

for simbolo in simbolos:
    texto=texto.replace(simbolo," " + simbolo + " ")

palabras_lista=texto.split()
palabras_lista.sort()
for palabra in palabras_lista:
    print(palabra)
```

Tokenización

forma básica

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

simbolos=["(",")",".",",",";",":","\\",""]

for simbolo in simbolos:
    texto=texto.replace(simbolo," " + simbolo + " ")

palabras_lista=texto.split()
palabras_lista.sort()
for palabra in palabras_lista:
    print(palabra)
```


Tokenización

- ▶ En el programa anterior vimos el uso de la función `replace()` de los textos, como una forma sencilla de separar símbolos.
- ▶ Sin embargo, podemos ver en la lista de palabras resultantes, que ahora se están separando cosas que sería mejor mantener unidas, como las siglas.
- ▶ Para lograr esto, es necesario una serie de reglas del estilo:
 - ▶ Si el símbolo está entre dos **letras mayúsculas**, no lo quites.
 - ▶ Si el símbolo está entre dos **números**, no lo quites.
- ▶ Pero cómo decirle a la computadora qué es una **letra mayúscula** o qué es un **número**.
- ▶ Existe una herramienta muy poderosa que es capaz de manejar el texto con grupos de letras (como mayúsculas, minúsculas, dígitos, etc) y reglas de repetición.

Expresiones Regulares

Hemos hablado sobre los problemas de reconocer cierto tipo de palabras o tokens. Sin embargo, son cosas que presentan ciertos patrones.

- ▶ Las siglas o abreviaturas, como: S.A. de C.V. o C.P.
- ▶ Precios y números, como: \$16.99 , 3.1416 \$25.
- ▶ Fechas, como: 15/Nov/1997 , 18/09/93.
- ▶ Correos electrónicos: gil@iingen.unam.mx

Hay muchas expresiones que siguen ciertos patrones, y es precisamente para detectar estas expresiones que nos pueden servir las expresiones regulares.

Una expresión regular es una formula en un lenguaje especial que se utiliza para especificar clases particulares de texto que siguen algún patrón.

Expresiones regulares

- ▶ Python soporta el uso de expresiones regulares, con ayuda de un módulo llamado `re`

```
import re
```

- ▶ Comenzaremos con las expresiones regulares más simples, que son aquellas que coinciden consigo misma, esto es equivalente a buscar un texto concreto.

Expresiones regulares

```
import re

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

expresion_regular=re.compile(r"México")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```

- En este ejemplo utilizamos la función `compile()` del módulo `re` para definir una expresión regular, en este caso solo usamos una palabra tal cual.

Expresiones regulares

- ▶ Este es el mismo ejemplo anterior, recortado para enfocarnos en la parte más importante.

```
expresion_regular=re.compile(r"México")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```

- ▶ El resultado de la función `compile()` lo almacenamos en la variable `expresion_regular`.
- ▶ Las expresiones regulares (como la que ahora se encuentra dentro de la variable `expresion_regular`) cuentan con funciones propias. Aquí estamos usando la función `search()`, que sirve para encontrar la expresión regular en un texto dado.
- ▶ El resultado de la búsqueda se almacena en la variable `resultado_busqueda` para mostrarlo en pantalla más adelante.

Expresiones regulares

Es importante notar dos detalles en este proceso:

- ▶ Hay una 'r' justo antes del texto, dentro de la definición de la expresión regular. Esto es para aclarar que se usará una expresión regular y que los símbolos especiales de Python no interfieran.
- ▶ Para mostrar el resultado, se usa: `group(0)`. Esto quiere decir que queremos el primer grupo (recuerden que los índices comienzan en 0) del resultado. Es posible hacer agrupaciones dentro de las expresiones regulares, lo veremos más adelante, por ahora, usen así la función para obtener el texto encontrado.

```
expresion_regular=re.compile(r"México")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```


Expresiones regulares

- ▶ Algo que tienen que considerar, es que la función `search()` va a regresar únicamente la primer coincidencia que encuentre, para encontrar todas, pueden usar la función `finditer()`

```
expresion_regular=re.compile(r"México")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

- ▶ Esta función tiene el mismo comportamiento que `search()` pero regresa todas las coincidencias, para entrar a cada una de ellas, es necesario usar un bucle para recorrer la variable como si fuera una lista. A esto se le llama un objeto iterable (si, las listas son iterables).

Expresiones regulares

- ▶ Comencemos ahora con los símbolos especiales de las expresiones regulares.
- ▶ Veamos el punto '.'. El punto coincide con un caracter, el que sea.

```
expresion_regular=re.compile(r".")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```

```
expresion_regular=re.compile(r".")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Sigamos ahora con el asterisco '*'.
- ▶ El asterisco permite cualquier cantidad de repeticiones del símbolo que lo precede.

```
expresion_regular=re.compile(r".*")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```

```
expresion_regular=re.compile(r".*")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- Acabamos de ver como funciona el asterisco junto con el punto, pero también funciona con letras, aquí les presento un ejemplo un tanto más confuso en su salida.

```
expresion_regular=re.compile(r"I*")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```

```
expresion_regular=re.compile(r"I*")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- Acabamos de ver como funciona el asterisco junto con el punto, pero también funciona con letras, aquí les presento un ejemplo un tanto más confuso en su salida.

```
expresion_regular=re.compile(r"I*")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```

```
expresion_regular=re.compile(r"I*")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Un efecto similar al asterisco lo logra el símbolo de suma '+'.
 - ▶ La diferencia radica en que la suma significa una o más repeticiones, no cero o más.

```
expresion_regular=re.compile(r"I+")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```

```
expresion_regular=re.compile(r"I+")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Pasemos al siguiente símbolo especial: '^'.
- ▶ Se utiliza para indicar el inicio del texto.

```
expresion_regular=re.compile(r"^.")  
resultado_busqueda=expresion_regular.search(texto)  
  
print(resultado_busqueda.group(0))
```

```
expresion_regular=re.compile(r"^.")  
resultados_busqueda=expresion_regular.finditer(texto)  
  
for resultado in resultados_busqueda:  
    print(resultado.group(0))
```

- ▶ Lean un archivo línea por línea.
- ▶ Para cada línea, busquen la expresión regular de la diapositiva anterior y muestren el resultado.

Expresiones regulares

- ▶ El siguiente símbolo especial: '\$'.
- ▶ Se utiliza para indicar el final del texto.

```
expresion_regular=re.compile(r".$")
resultado_busqueda=expresion_regular.search(texto)

print(resultado_busqueda.group(0))
```

```
expresion_regular=re.compile(r".$")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Este es muy fácil ya.

- ▶ Hagan lo mismo que en el ejercicio pasado, pero para la expresión regular de la diapositiva anterior.

Expresiones regulares

- ▶ El siguiente símbolo especial: '?'.
 - ▶ Es parecido al símbolo '+'. Pero en lugar de ser 1 o más, es 1 o 0. Acepta que aparezca o la letra anterior.
 - ▶ Esta vez solo usaremos el ejemplo iterativo, ya sabemos que el otro solo muestra el primer resultado del iterativo.

```
expresion_regular=re.compile(r"artículos?")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ El signo de interrogación (?) tiene además una función importante.
- ▶ Se puede usar en conjunto con los símbolos que buscan muchas letras (como el '+' y el '*').
- ▶ Normalmente la expresión regular busca el texto mas largo que coincida, pero si se usa el signo de interrogación eso cambiará y buscará el mas corto.

```
expresion_larga=re.compile(r"\(.*\)")
expresion_corta=re.compile(r"\(..*?\)")
resultados_largos=expresion_larga.finditer(texto)
resultados_cortos=expresion_corta.finditer(texto)

for resultado in resultados_largos:
    print(resultado.group(0))
print()
for resultado in resultados_cortos:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Seguramente han notado en los ejemplos anteriores que los símbolos afectan únicamente a la letra que los precede.
- ▶ Es posible también afectar a un grupo de letras, para eso, es necesario agruparlas, y ese es precisamente la función de los paréntesis: "()".

```
expresion_regular=re.compile(r"(el)?(los)? artículos?")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ En el ejemplo anterior se logra un efecto similar a capturar una de las dos expresiones "el" o "los", seguido de "artículo" o "artículos".
- ▶ Las expresiones regulares tienen una instrucción propia para especificar que se quiere una cosa o la otra, y es con el símbolo: '|'

```
expresion_regular=re.compile(r"(el|los) artículos?")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Otro símbolo especial, los corchetes: "[]".
- ▶ Lo podemos ver como un atajo para poner diferentes opciones, aunque las opciones solo pueden ser de una letra, aunque se pueden poner muchas letras dentro del corchete. Al momento de la búsqueda el programa intentará coincidir con alguna de ellas.

```
expresion_regular=re.compile(r"M[eéa] [xr] ic? [ao] (nos)?")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Además de eso, los corchetes también aceptan rangos de valores. Para eso se utilizan los extremos del rango con un símbolo de menos (-) entre ellos.
- ▶ Los usos del corchete se pueden combinar.

```
expresion_regular=re.compile(r"M[a-zA-ú] [a-z]ic?[a-z] (nos)?")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```


Expresiones regulares

- ▶ Y todavía un uso más del corchete. Podemos usarlo para indicar lo que queremos que NO coincida en nuestra expresión regular.
- ▶ Para lograr eso, usamos el símbolo '^' justo al inicio del corchete, antes del conjunto de letras que queremos excluir.

```
expresion_regular=re.compile(r"M[^a-z][^0-9]ic?[a-z](nos)?")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Con los corchetes podemos hacer recorridos de letras o números para manejar grupos.
- ▶ Pero las expresiones regulares tienen predefinidos ciertos grupos de letras, números y símbolos que facilitan las cosas aún más.
- ▶ Comencemos con el grupo: `\d` . Que incluye a todos los dígitos. Sería equivalente a: `[0-9]`

```
expresion_regular=re.compile(r"\d+(,\d+)*(\.\d+)?")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Otro grupo predefinido es el de los espacios: `\s`
- ▶ Este conjunto incluye al espacio, al tabulador, y a otros símbolos que se usan para poner espacios en blanco en un texto.

```
expresion_regular=re.compile(r"[\s]+")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

Expresiones regulares

- ▶ Un grupo muy importante también: `\w`
- ▶ Este conjunto incluye todos los caracteres que se esperaría pudieran aparecer en una palabra. Esto incluye a todas las letras, todos los dígitos y el guión bajo (`'_'`)
- ▶ Ojo, este grupo tiene la ventaja de que SI incluye a los acentos.

```
expresion_regular=re.compile(r"\w+")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
```

- ▶ Los grupos en mayúsculas, es decir: `\D` , `\S` , y `\W` .
Coinciden con los opuestos de `\d` , `\s` y `\w` respectivamente.
- ▶ Es decir, es equivalente a ponerlos dentro de corchetes con '^' inicial.

Obtengan con una expresión regular los artículos constitucionales que se mencionan en el texto. Tomen en cuenta que:

- ▶ Los artículos van precedidos de la palabra: "artículo" o "artículos".
- ▶ Los artículos son números.
- ▶ Cuando se habla de varios artículos, éstos pueden estar separados por comas, o por la palabra "y"

Expresiones regulares

Por último hablemos de grupos y referencias.

- ▶ Los paréntesis son capaces de agrupar expresiones para usar instrucciones sobre más de una letra, cierto. Pero también se acuerdan del grupo que formaron, los cuáles se van enumerando.
- ▶ El primer paréntesis que se usa, forma al grupo \1, el segundo forma al grupo \2 y así sucesivamente.

```
expresion_regular=re.compile(r"\(([A-Z]\w*)\)ate.*\1")
resultados_busqueda=expresion_regular.finditer(texto)

for resultado in resultados_busqueda:
    print(resultado.group(0))
    print("\n",resultado.group(1),"\n")
```

Expresiones regulares

- ▶ Las expresiones regulares de python (el módulo `re`) cuentan con funciones bastante útiles además de la búsqueda.
- ▶ Una de ellas es la sustitución. Para lo que se usa la función: `sub()`

```
expresion_regular=re.compile(r"([^\w\s][^A-Z\d])")
texto_nuevo=expresion_regular.sub(r" \1",texto)

print(texto_nuevo)
```


Expresiones regulares

- ▶ Las expresiones regulares de python (el módulo `re`) cuentan con funciones bastante útiles además de la búsqueda.
- ▶ Una de ellas es la sustitución. Para lo que se usa la función: `sub()`

```
expresion_regular=re.compile(r"([^\w\s][^A-Z\d])")
texto_nuevo=expresion_regular.sub(r" \1",texto)

print(texto_nuevo)
```

El último programa separa de las palabras los signos de puntuación que ocurren de su lado derecho.

- ▶ Usen el texto de salida anterior y un nuevo paso de proceso para separar los signos de puntuación izquierdos, de modo que queden las palabras y los signos completamente separados por espacios.

- Esta vez también les presentaré aquí la solución del ejercicio.

```
expresion_derecha=re.compile(r"([^\w\s][^\A-Z\d])")
texto_nuevo=expresion_derecha.sub(r" \1",texto)
expresion_izquierda=re.compile(r"([^\A-Z\d][^\w\s])")
texto_ultimo=expresion_izquierda.sub(r"\1 ",texto_nuevo)

print(texto_ultimo)
```

- Ya que usaremos ese nuevo texto con todo bien separado.

Expresiones Regulares

- ▶ ¿Recuerdan nuestro tokenizador básico?
- ▶ Tenía algunas fallas, pero ahora que las cosas están bien separadas por espacios, podemos lograr algo mejor.

```
expresion_derecha=re.compile(r"([^\w\s][^\A-Z\d])")
texto_nuevo=expresion_derecha.sub(r" \1",texto)
expresion_izquierda=re.compile(r"([^\A-Z\d][^\w\s])")
texto_ultimo=expresion_izquierda.sub(r"\1 ",texto_nuevo)

tokens_lista=texto_ultimo.split()

for token in tokens_lista:
    print(token)
```

- ▶ Esta nueva versión de tokenizador funciona mejor gracias a las expresiones regulares.
- ▶ Aún le faltan cosas para ser del todo confiable. Pero por ahora podemos ver cómo se van formando muchas de las reglas que componen un tokenizador completo, y además es importante conocer las expresiones regulares como herramienta.
- ▶ Por supuesto, los tokenizadores, al igual que otras herramientas del PLN, están bastante estudiados y los hay a nuestra disposición sin tener que armar uno nosotros mismos.

NLTK

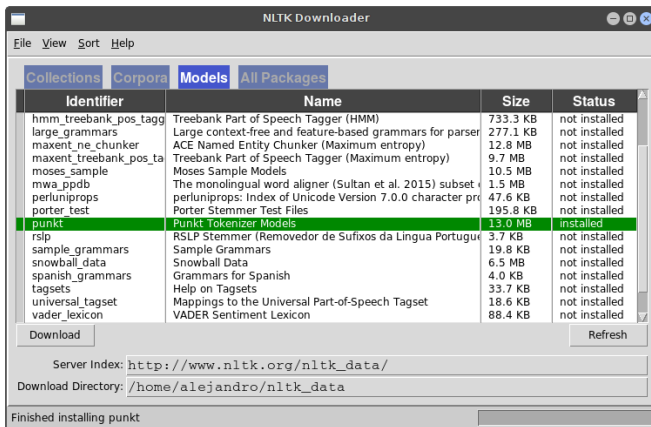
- ▶ NLTK (Natural language tool kit) es un módulo de Python que contiene herramientas para el manejo del lenguaje natural.
- ▶ Para usarlas, al igual que con los otros módulos, se tiene que importar.

```
import nltk
```

- ▶ Como seguramente ya se imaginarán, NLTK tiene su tokenizador. Así que comencemos por allí.
- ▶ Sin embargo, NLTK tiene tantas cosas, que las guarda en la nube para que cada solo se descarguen las herramientas que cada quien necesita.
- ▶ Podemos abrir el "navegador" de NLTK desde el mismo Python. ¿Recuerdan cómo usar la consola de Python? si no, lo pueden hacer también desde el editor:

```
import nltk
```

```
nltk.download()
```

- ▶ Verán una pantalla como esta, aquí está todo el contenido de nltk para descargar.
- ▶ Vayan a la sección de **Models** y descarguen **punkt**, que como verán, son modelos de tokenización.

- ▶ Y ahora que ya descargamos el tokenizador, podemos cerrar esa ventana y usarlo. De regreso en Python.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")

for token in tokens:
    print(token)
```

- ▶ Y ahora que ya descargamos el tokenizador, podemos cerrar esa ventana y usarlo. De regreso en Python.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")

for token in tokens:
    print(token)
```

Conteo de palabras

o mejor dicho de tokens

- ▶ Así que ahora tenemos dos maneras de obtener una lista de tokens para nuestros textos.
- ▶ Veamos un par de usos, pueden ocupar la lista que más les guste.
- ▶ Para empezar, podemos usarlas para contar el total de palabras del documento.

```
# Aquí, nuestra lista de tokens se llama "tokens"
```

```
palabras_total=len(tokens)
```

```
print(palabras_total)
```

Palabras diferentes

- ▶ Además de las palabras totales, nos puede interesar conocer las palabras diferentes que utiliza un texto.
- ▶ Para esto, podemos utilizar el `set` de Python.
- ▶ Un `set` es un conjunto. Es similar a una lista pero están pensados para hacer operaciones entre conjuntos (como uniones, intersecciones o diferencias). Y una característica particular que por ahora nos interesa bastante es que sus elementos no se repiten.
- ▶ Se usa la función `set()` para convertir una lista en un conjunto.

```
# Aquí, nuestra lista de tokens se llama "tokens"
```

```
tokens_conjunto=set(tokens)
```

```
palabras_diferentes=len(tokens_conjunto)
```

```
print(palabras_diferentes)
```

Riqueza léxica

- ▶ La riqueza léxica es la relación que existe entre la extensión de un texto y el número de palabras distintas que contiene.
- ▶ Así que ahora es muy fácil para nosotros calcularla.

```
# Aquí, nuestra lista de tokens se llama "tokens"  
tokens_conjunto=set(tokens)  
  
palabras_totales=len(tokens)  
palabras_diferentes=len(tokens_conjunto)  
  
riqueza_lexica=palabras_diferentes/palabras_totales  
  
print(riqueza_lexica)
```

Funciones

en Python claro

- ▶ Hemos visto muchas funciones. Ahora veamos que también podemos crearlas nosotros mismos.
- ▶ Hagamos una función que calcule la riqueza léxica si le damos una lista de tokens.

```
def riqueza_lexica(tokens):  
    tokens_conjunto=set(tokens)  
  
    palabras_totales=len(tokens)  
    palabras_diferentes=len(tokens_conjunto)  
  
    riqueza_lexica=palabras_diferentes/palabras_totales  
  
    return riqueza_lexica
```

Funciones

```
import nltk

def riqueza_lexica(tokens):
    tokens_conjunto=set(tokens)
    palabras_totales=len(tokens)
    palabras_diferentes=len(tokens_conjunto)
    riqueza_lexica=palabras_diferentes/palabras_totales
    return riqueza_lexica

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")
riqueza_lexica=riqueza_lexica(tokens)
print(riqueza_lexica)
```


Funciones

```
import nltk

def riqueza_lexica(tokens):
    tokens_conjunto=set(tokens)
    palabras_totales=len(tokens)
    palabras_diferentes=len(tokens_conjunto)
    riqueza_lexica=palabras_diferentes/palabras_totales
    return riqueza_lexica

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")
riqueza_lexica=riqueza_lexica(tokens)
print(riqueza_lexica)
```

- ▶ Modificar la función que se creó para que en lugar de recibir una lista de tokens, reciba texto (el texto que se lea de un archivo por ejemplo).
- ▶ NOTA: Se tendrán que agregar todos los pasos necesarios para que la función trabaje de manera correcta y que devuelva la riqueza léxica del texto que introduzcan.

Conteo individual

- ▶ También podemos usar la lista de tokens para hacer conteos de palabras individuales.
- ▶ Para esto, utilizamos la función `count()` de la lista.

```
# Aquí, nuestra lista de tokens se llama "tokens"  
  
conteo_individual=tokens.count("el")  
  
print(conteo_individual)  
  
palabras_totales=len(tokens)  
porcentaje=100*conteo_individual/palabras_totales  
  
print(porcentaje, "%")
```

Funciones

- ▶ Algunas funciones reciben más de un dato. Cuando esto pasa se separan por comas dentro de los paréntesis.
- ▶ Cuando definimos funciones, también podemos definir más de un parámetro de entrada. De igual manera, separados por comas.

```
def porcentaje(palabra,texto):  
    # Aquí va el programa  
    # Se pueden usar las variables "palabra" y "texto"  
    # al momento de usar la función, los parámetros se  
    # asignan por el orden. Es decir, el primero  
    # parámetro será "palabra", el segundo "texto".  
    # Pueden definir los parametros como quieran  
    # Solo asegurense de poner el orden correcto de los  
    # datos al momento de usar su función
```

Para que quede bien claro el uso de funciones

- ▶ Definir una función que calcule el porcentaje que ocupa una palabra dentro de un textos. Usen como parámetros de entrada la palabra y el texto.

- ▶ De regreso a NLTK, también podemos usar los tokens para crear una variable de tipo **Text** de NLTK.
- ▶ Estos objetos tienen varias funciones útiles.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")

texto_nltk=nltk.Text(tokens)
```

NLTK

Concordancias

- ▶ Las concordancias muestran todas las apariciones de una palabra junto con algo del texto que la rodea.
- ▶ Con la función `concordance()` de el `Text` de NLTK es mostrar las concordancias de una palabra.

```
import nltk
```

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"  
archivo_nombre="P_IFT_290216_73_Acc.txt"
```

```
with open(carpeta_nombre+archivo_nombre,"r") as archivo:  
    texto=archivo.read()
```

```
tokens=nltk.word_tokenize(texto,"spanish")
```

```
texto_nltk=nltk.Text(tokens)  
texto_nltk.concordance("artículo")
```

RE

Concordancias

- ▶ Aprovecharé este punto para explicar un símbolo de las expresiones regulares que no había mencionado. Los corchetes: {}
- ▶ Y para mostrarles cómo hacer concordancias con RE, por si quieren ampliar su funcionamiento o simplemente no cargar nltk.

```
import re
```

```
# Aqui va la lectura del archivo, por ahora la estoy omitiendo
```

```
expresion=re.compile(r".{,30}[\s~][Aa]rtículos? .{,30}")  
resultados_busqueda=expresion.finditer(texto)
```

```
for resultado in resultados_busqueda:  
    print(resultado.group(0))
```


NLTK

Palabras similares

- ▶ De nuevo en NLTK, veamos otra de las funciones que tiene sus **Text**.
- ▶ Hemos visto que la función **concordance()** nos muestra una palabra en su contexto.
- ▶ Pero también tiene la función **similar()** que es capaz de mostrarnos otras palabras, que tengan contextos similares.

```
import nltk
```

```
# Aquí va la lectura del archivo, por ahora la estoy omitiendo
```

```
tokens=nltk.word_tokenize(texto,"spanish")
```

```
texto_nltk=nltk.Text(tokens)
```

```
texto_nltk.similar("artículo")
```

NLTK

Palabras similares

- ▶ Para este tipo de funciones, entre más texto se tenga para hacer el análisis es mejor.
- ▶ Hasta ahora, hemos usado un texto corto para los ejemplos, probemos ahora con el conjunto de todos los que tenemos.

```
import nltk
```

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"  
archivo_nombre="DOF_P_IFT_291116_672_Acc.txt"
```

```
with open(carpeta_nombre+archivo_nombre,"r") as archivo:  
    texto=archivo.read()
```

```
tokens=nltk.word_tokenize(texto,"spanish")
```

```
texto_nltk=nltk.Text(tokens)  
texto_nltk.similar("artículo")
```

- Por supuesto, podemos ver cuál es el contexto que comparten las palabras similares.

```
tokens=nltk.word_tokenize(texto,"spanish")

texto_nltk=nltk.Text(tokens)
texto_nltk.similar("artículo")
print()
texto_nltk.common_contexts(["artículo","instituto"])
```

- La función `print()` es para separar los resultados y sea más claro el contenido de cada parte.

- ▶ Otra función muy interesante es `dispersion_plot()`.
- ▶ Esta función muestra una gráfica con la aparición de una lista de palabras buscadas a lo largo de todo el texto.

```
tokens=nltk.word_tokenize(texto,"spanish")
texto_nltk=nltk.Text(tokens)

lista_palabras=["Instituto","Ley","Elija","ley"]
texto_nltk.dispersion_plot(lista_palabras)
```

NLTK

Distribución de frecuencias

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

tokens=nltk.word_tokenize(texto,"spanish")

texto_nltk=nltk.Text(tokens)

distribucion=nltk.FreqDist(texto_nltk)

lista_frecuencias=distribucion.most_common()
print(lista_frecuencias)
```

- ▶ De la distribución de frecuencias también podemos obtener la frecuencia de una palabra en particular.
- ▶ Como podrán ver, esto se logra de manera similar a los índices de una lista, pero en lugar de el índice (el número que indica la posición dentro de la lista) se usa la palabra misma, como texto.

A esta altura ya tenemos la lista de tokens en "tokens"

```
texto_nltk=nltk.Text(tokens)
```

```
distribucion=nltk.FreqDist(texto_nltk)
```

```
print(distribucion["Instituto"])
```

Diccionarios

en Python

- ▶ Esas "listas" que en lugar de usar índices usan palabras, se llaman diccionarios. Es otra herramienta que tiene Python.
- ▶ OJO, el resultado de la función `FreqDist()` de NLTK en realidad no es un diccionario, ya que tiene funciones propias de NLTK; pero se comporta como un diccionario para obtener su contenido usando palabras.

```
info={"nombre":"Mi nombre","apellido":"Mi apellido"}
info["edad"]=100
info["curso"]="Python"

for dato in info:
    print(dato,":",info[dato])
```

- ▶ Muchas veces nos interesa observar de una manera visual cómo cambia la frecuencia de las palabras.
- ▶ Las distribuciones de NLTK tienen opciones para obtener gráficas.

A esta altura ya tenemos la lista de tokens en "tokens"

```
texto_nltk=nltk.Text(tokens)
distribucion=nltk.FreqDist(texto_nltk)

distribucion.plot()
```


- ▶ Seguramente su gráfica muestra demasiadas palabras como para que se entienda claramente el valor de cada una.
- ▶ Se puede disminuir el número de palabras que se muestran, pueden dar el número que desean como parámetro a la función.

A esta altura ya tenemos la lista de tokens en "tokens"

```
texto_nltk=nltk.Text(tokens)
distribucion=nltk.FreqDist(texto_nltk)

distribucion.plot(40)
```

- ▶ Una nota, es probable que su gráfica muestre las palabras cortadas en el eje horizontal.
- ▶ Esto se puede corregir si agregan las siguientes líneas en algún punto ANTERIOR a usar la gráfica:

```
from matplotlib import rcParams  
rcParams.update({"figure.autolayout": True})
```

- ▶ No se preocupen mucho por este código, son líneas especiales de configuración de las gráficas.

- ▶ Las distribuciones de NLTK también cuentan con una función que regresa los *hapaxes* de un texto. Para ello usamos la función `hapaxes()` de la distribución, y obtenemos una lista con las palabras.
- ▶ Un *hapax* es una palabra que aparece únicamente una vez en el texto.

A esta altura ya tenemos la lista de tokens en "tokens"

```
texto_nltk=nltk.Text(tokens)
distribucion=nltk.FreqDist(texto_nltk)
hapaxes=distribucion.hapaxes()
```

```
for hapax in hapaxes:
    print(hapax)
```

- ▶ La gráfica de la distribución tiene un parámetro opcional para transformar la gráfica en una acumulativa.
- ▶ Verán que es un parámetro raro, ya que es como poner y asignar una variable dentro de los paréntesis, y básicamente es eso justamente lo que se hace. Ese parámetro no tiene lugar fijo, como se puede ver en el ejemplo.
- ▶ Además notarán que se usa la palabra especial **True**, que es, como su nombre lo dice, el valor "verdadero". Este valor se utiliza para cuando se quiere valores binarios (como cuando se revisa un **if**), también existe el valor **False**.

```
distribucion.plot(cumulative=True)  
distribucion.plot(40,cumulative=True)
```

NLTK

Gráfica acumulativa de frecuencias

- ▶ Podemos comparar el total de palabras con el aumento acumulativo de la gráfica.
- ▶ Observen como las primeras 40 palabras más usadas cuentan por más de la mitad del número total de palabras (que son más de 300)

A esta altura ya tenemos la lista de tokens en "tokens"

```
tokens_conjunto=set(tokens)
palabras_totales=len(tokens)
palabras_diferentes=len(tokens_conjunto)
```

```
texto_nltk=nltk.Text(tokens)
distribucion=nltk.FreqDist(texto_nltk)
```

```
print(palabras_totales)
print(palabras_diferentes)
```

```
distribucion.plot(cumulative=True)
distribucion.plot(40,cumulative=True)
```

- ▶ Algo que pueden notar de las gráficas de distribuciones, y que sin duda ya esperaban. Es que las palabras más frecuentes, son palabras como "el", "de", "la", etc. A este tipo de palabras, se le llaman palabras funcionales.
- ▶ En todos los archivos que analicen siempre estas palabras ocuparán los lugares mas frecuentes. Es por esta razón que buscar las palabras más frecuentes en realidad no da información sobre el contenido de un archivo.
- ▶ Afortunadamente, estas palabras son muy conocidas y estudiadas, y ya que todo el mundo por lo regular quiere quitarlas, no es difícil encontrar listas enumerándolas.
- ▶ NLTK cuenta con su propia lista de estas palabras, también llamadas *stopwords*.

NLTK

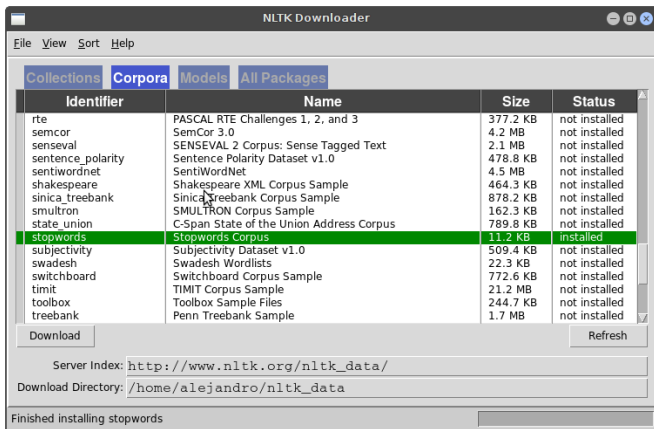
Palabras funcionales

- ▶ Para obtenerla, hay que descargarla.
- ▶ Recuerden que NLTK tiene su propio sistema de descarga que se usa desde Python.

```
import nltk  
  
nltk.download()
```

NLTK

Palabras funcionales



- Una vez que aparezca la pantalla de descargas vayan a la sección de **Corpora** y descarguen **stopwords**, que como verán, son listas de palabras funcionales en varios idiomas.

- ▶ Y ahora que ya descargamos las listas, podemos cerrar esa ventana.
- ▶ A partir de ahora podemos usar las listas de *stopwords* de NLTK. Veamos que palabras tiene para el español.

```
import nltk

palabras_funcionales=nltk.corpus.stopwords.words("spanish")

for palabras_funcional in palabras_funcionales:
    print(palabras_funcional)
```

NLTK

Palabras funcionales

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

palabras_funcionales=nltk.corpus.stopwords.words("spanish")

tokens=nltk.word_tokenize(texto,"spanish")

tokens_limpios=[]
for token in tokens:
    if token not in palabras_funcionales:
        tokens_limpios.append(token)

print(len(tokens))
print(len(tokens_limpios))
```

NLTK

Palabras funcionales

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

palabras_funcionales=nltk.corpus.stopwords.words("spanish")

tokens=nltk.word_tokenize(texto,"spanish")

tokens_limpios=[]
for token in tokens:
    if token not in palabras_funcionales:
        tokens_limpios.append(token)

print(len(tokens))
print(len(tokens_limpios))
```

- ▶ Podemos también obtener las gráficas de los nuevos datos sin palabras funcionales.
- ▶ Por cierto, quizá notaron en el ejemplo anterior que mostramos en pantalla (con `print()`) el resultado directo de una función, sin asignarlo antes a una variable. En Python podemos hacer poner una cadena de funciones y usar los resultados inmediatamente, incluso podemos hacer algo tan drástico como el siguiente ejemplo, pero por norma general, no se recomienda, se vuelve poco legible y no se pueden recuperar los datos intermedios.

```
# Después del ejemplo anterior...
```

```
nltk.FreqDist(nltk.Text(tokens_limpios)).plot(40)
```

- ▶ Como pueden observar, aún hay elementos muy frecuentes que sería útil quitar, como los signos de puntuación.
- ▶ Usaré este punto para explicarles un detalle sobre el texto que no había mencionado antes. Y es que se puede comportar un poco como una lista de letras.
- ▶ El ejemplo siguiente es una solución ingenua para quitar los signos de puntuación, es decir, es muy simple, pero no es la solución ideal ya que esto quitará más cosas solamente los signos de puntuación, pero depende del objetivo final, puede que la pérdida no sea importante.

NLTK

Palabras funcionales

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre,"r") as archivo:
    texto=archivo.read()

palabras_funcionales=nltk.corpus.stopwords.words("spanish")

tokens=nltk.word_tokenize(texto,"spanish")

tokens_limpios=[]
for token in tokens:
    if token not in palabras_funcionales:
        if len(token) > 1:
            tokens_limpios.append(token)

nltk.FreqDist(nltk.Text(tokens_limpios)).plot(40)
```

- ▶ Del ejemplo anterior, podemos ver como las palabras "limpias" más comunes, ya nos dan información importante sobre un texto.
- ▶ Por cierto, para ustedes que no tienen que ahorrar espacio, recuerden que esto no es recomendable:

```
nltk.FreqDist(nltk.Text(tokens_limpios)).plot(40)
```

- ▶ Es mejor esto:

```
texto_limpio_nltk=nlk.Text(tokens_limpios)
distribucion_limpia=nlk.FreqDist(texto_limpio_nltk)

distribucion_limpia.plot(40)
```

- ▶ Las colocaciones son secuencias de palabras que ocurren juntas de una forma inusualmente frecuente.
- ▶ NLTK tiene una función particular para obtener las colocaciones de un texto.

```
# Aquí ya tenemos tokens y tokens_limpios
```

```
texto_nltk=nltk.Text(tokens)
```

```
texto_limpio_nltk=nltk.Text(tokens_limpios)
```

```
texto_nltk.collocations()
```

```
print()
```

```
texto_limpio_nltk.collocations()
```

- ▶ Se puede usar tanto el texto completo como el "limpio", ambos dan información que puede ser útil, como "rubro citado" o "S.A. C.V." "Ciudad México". Que aparecen solo en uno de los dos.

- ▶ Un recurso muy útil para el PLN es el etiquetado PoS (de *Part of Speech*).
- ▶ Con esta herramienta, la computadora hace su mejor esfuerzo por asignarle a cada palabra de un texto la parte de la oración que le corresponde (sustantivo, verbo, adjetivo, determinante, etc.)
- ▶ NLTK cuenta con un etiquetador PoS. Desafortunadamente, es solo para inglés.
- ▶ Afortunadamente, también tiene acceso a un etiquetador externo (de Stanford) que si es capaz de manejar el español.
- ▶ Esta herramienta no la tiene por defecto, y al ser externa tampoco se descarga desde el sistema de descarga que hemos estado usando hasta ahora.
- ▶ <https://nlp.stanford.edu/software/tagger.shtml>

- En la página de Stanford podemos encontrar toda la información sobre el etiquetador y ligas de descarga.



The Stanford Natural Language Processing Group

[people](#) [publications](#) [research blog](#) [software](#) [teaching](#) [local](#)

Software > Stanford Log-linear Part-Of-Speech Tagger

Stanford Log-linear Part-Of-Speech Tagger

[About](#) | [Questions](#) | [Mailing lists](#) | [Download](#) | [Extensions](#) | [Release history](#) | [FAQ](#)

About

A Part-Of-Speech Tagger (POS Tagger) is a piece of software that reads text in some language and assigns parts of speech to each word (and other token), such as noun, verb, adjective, etc., although generally computational applications use more fine-grained POS tags like 'noun-plural'. This software is a Java implementation of the log-linear part-of-speech taggers described in these papers (if citing just one paper, cite the 2003 one):

- ▶ La descarga básica de su etiquetador también funciona solo con inglés, así que necesitamos la descarga completa, es la segunda liga.
- ▶ Esto va a descargar un archivo comprimido, es importante que lo descompriman y que recuerden donde lo colocan. Recuerden la forma de obtener rutas completas, será necesario para usarlo.

Download

[Download basic English Stanford Tagger version 3.8.0 \[25 MB\]](#)

[Download full Stanford Tagger version 3.8.0 \[129 MB\]](#)

The basic download is a 24 MB zipped file with support for tagging English. The full download is a 124 MB zipped file, which includes additional English models and trained models for Arabic, Chinese, French, Spanish, and German. In both cases most of the file size is due to the trained model files. The only difference between the two downloads is the number of trained models included. If you unpack the tar file, you should have everything needed. This software provides a GUI demo, a command-line interface, and an API. Simple scripts are included to invoke the tagger. For more information on use, see the included README.txt.

- ▶ Una vez que hayan descargado y descomprimido el zip, hay dos archivos que deben localizar.
- ▶ El primero está directamente dentro de la carpeta que acaban de descomprimir, se llama `stanford-postagger.jar`.
- ▶ El segundo, está dentro de la carpeta `models`, y su nombre es `spanish.tagger` sin nada más.
- ▶ Para ambos archivos van a necesitar la ruta completa para usarlos con Python y NLTK.

- La función `StanfordPOSTagger()` recibe de parámetros los archivos que obtuvimos, primero el `spanish.tagger`, luego el `stanford-postagger.jar`.

```
from nltk.tag import StanfordPOSTagger

# Aquí obtenemos la lista de tokens en "tokens"

tagger="C:\\Users\\user\\Downloads\\...\\spanish.tagger"
jar="C:\\Users\\user\\Downloads\\...\\stanford-postagger.jar"

etiquetador=StanfordPOSTagger(tagger,jar)
etiquetas=etiquetador.tag(tokens)

for etiqueta in etiquetas:
    print(etiqueta)
```

- Con esa información, el `StanfordPOSTagger()` crea un etiquetador que luego podemos utilizar para etiquetar nuestra lista de tokens. En este caso, es mejor utilizar la lista que no está "limpia".

```
from nltk.tag import StanfordPOSTagger

# Aquí obtenemos la lista de tokens en "tokens"

tagger="C:\\Users\\user\\Downloads\\...\\spanish.tagger"
jar="C:\\Users\\user\\Downloads\\...\\stanford-postagger.jar"

etiquetador=StanfordPOSTagger(tagger,jar)
etiquetas=etiquetador.tag(tokens)

for etiqueta in etiquetas:
    print(etiqueta)
```

- ¿Y esas etiquetas qué?
- Cada etiqueta tiene un significado, en la página <https://nlp.stanford.edu/software/spanish-faq.shtml#tagset> pueden encontrar las etiquetas con sus significados y ejemplos.

6. What POS tag set does the parser use?

We use a simplified version of the tagset used in the AnCor 3.0 corpus / DEFT Spanish Treebank. The default AnCor tagset has hundreds of different extremely precise tags. This may be useful for some linguistic applications, but did not bode well for even a state-of-the-art part-of-speech tagger. We reduced the tagset to *85 tags*, a more manageable size that still allows for a useful amount of precision.

The tags are designed to remain compatible with the [FAGLES standard](#). In our tags, we simply null out most of the fields (using a label 0) that are not relevant for our purposes. The resulting compressed tagset is listed below.

Tag	Description	Example(s)
Adjectives		
ao0000	Adjective (ordinal)	<i>primera, segundo, últimos</i>
aq0000	Adjective (descriptive)	<i>populares, elegido, emocionada, andaluz</i>
Conjunctions		
cc	Conjunction (coordinating)	<i>y, o, pero</i>
cs	Conjunction (subordinating)	<i>que, como, mientras</i>
Determiners		
da0000	Article (definite)	<i>el, la, los, las</i>
dd0000	Demonstrative	<i>este, esta, esos</i>
de0000	"Exclamative" (TODO)	<i>qué (¡Qué pobre!)</i>
di0000	Article (indefinite)	<i>un, muchos, todos, otros</i>
dn0000	Numeral	<i>tres, doscientas</i>
do0000	Numeral (ordinal)	<i>el 65 aniversario</i>
dp0000	Possessive	<i>sus, mi</i>
dt0000	Interrogative	<i>cuántos, qué, cuál</i>