

# PLN con Python

Alejandro Pimentel

# Introducción

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

# PLN con Python

└─ Introducción

└─ Herramientas

└─ Herramientas

- Intérprete Python
- Editor de texto
- Internet

- En este curso utilizaremos en particular el intérprete de Python 3.
- En cuanto al editor, cualquier editor de texto será suficiente, desde el bloc de notas hasta cualquier IDE profesional con el que se sientan cómodos.
- Además, bueno, me parece que el internet es un requerimiento ya para todo, pero en este curso en particular descargaremos herramientas que se necesitan para trabajar, python incluido, si es que no lo tienen ya.



ANACONDA®



spyder

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

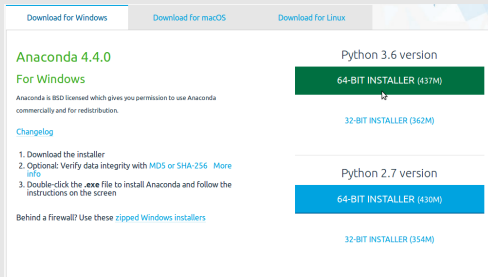
2017-07-27

# PLN con Python

- └─ Introducción
- └─ Herramientas
- └─ Herramientas

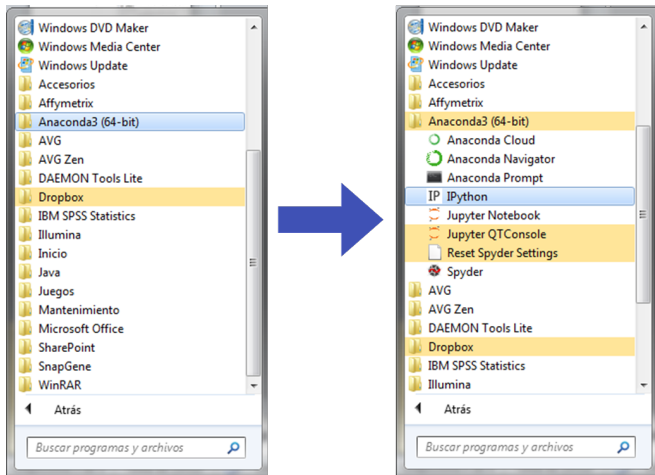


- Para aquellos que no han tenido experiencia con algún lenguaje de programación o python antes, les recomiendo que utilizen *Anaconda* (<https://www.continuum.io/downloads>), es un paquete que incluye el intérprete de python, y un editor diseñado específicamente para python (spyder).
- En la página de descargas, busquen la versión de anaconda para python3 y para la arquitectura de su computadora (64 o 32 bits, lo más probable es que sea de 64 bits).



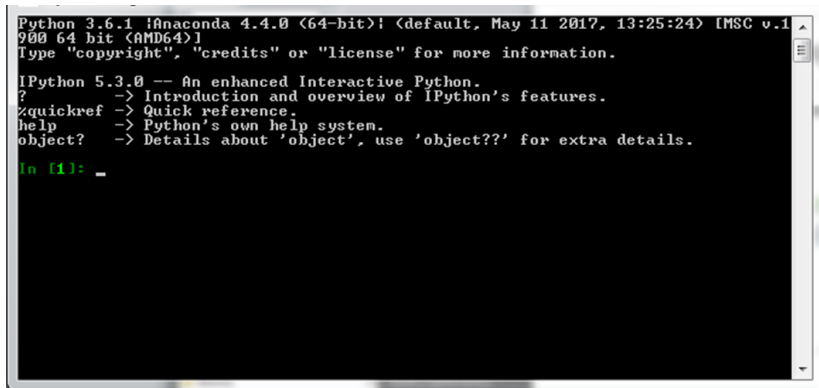
# Primero lo primero

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.

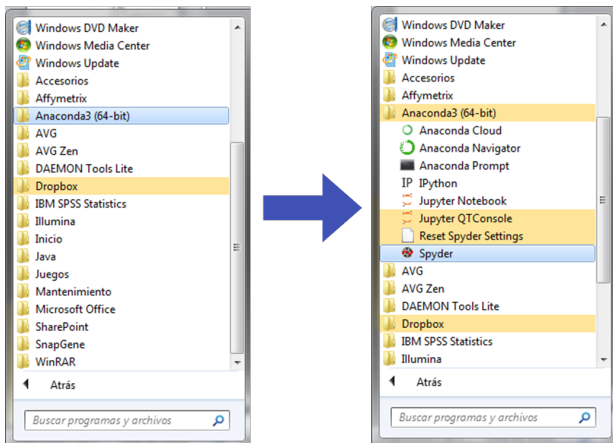
```
print("Hola Mundo")
```

- ▶ El mensaje debe ir entre comillas.

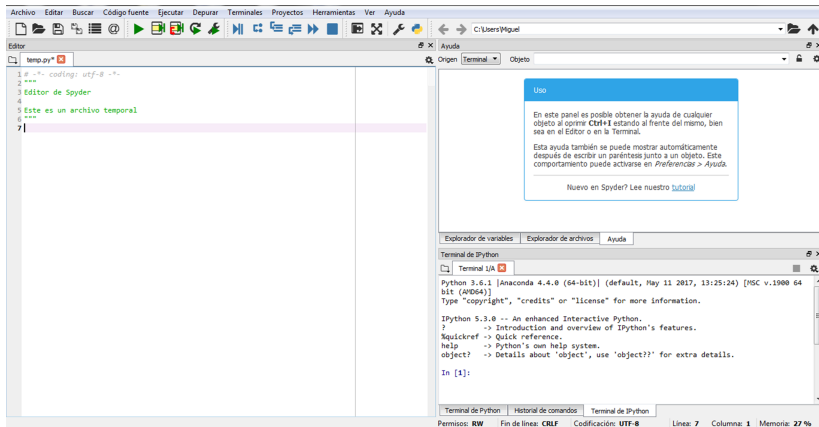
# El editor

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*.



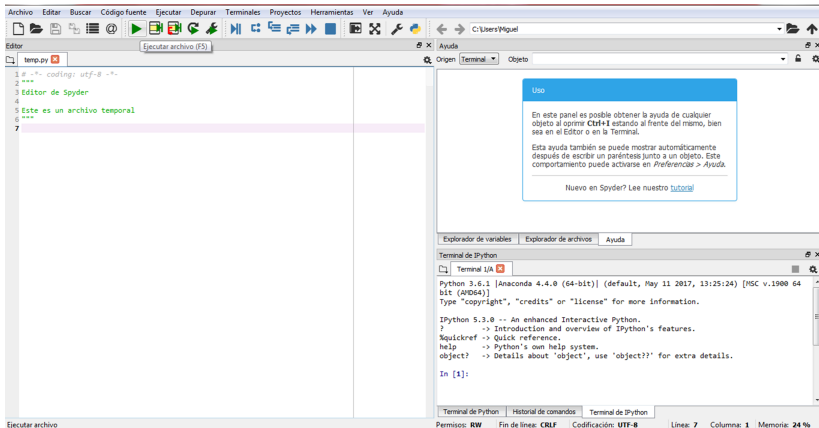
- ▶ Spyder cuenta con una consola en la que se pueden ejecutar comandos, o lo que se esté escribiendo en el editor.
- ▶ Además, cuenta con un apartado de ayuda para las funciones de Python.



- ▶ 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.



- ▶ 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)
```

# PLN con Python

└─ Introducción

└─ Operaciones

└─ Operaciones

► 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)
```

- Para el primer ejemplo de esta diapositiva, notarán que el resultado es 22, esto es debido a que, cuando se está trabajando texto (entre comillas) la operación + es una operación de concatenación. Es decir, junta en texto.
- Para el segundo ejemplo, se mostrará en pantalla el texto de la operación, no se evaluará nada, al poner comillas, esos ya no son números ni operaciones, es texto.
- El último ejemplo provocará un error, Python no sabe hacer operaciones entre texto y números.



# 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.

- 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.

- La función `print()` puede recibir varias cosas para mostrar en la pantalla, todas ellas separadas por coma, de esa manera, se separan las entradas.
- En el ejemplo estamos mandando tanto texto, como una operación, cada argumento (separado por coma) que mandamos, se evaluará y luego se mostrará en la pantalla.
- Si omitimos la coma, se producirá un error, ya que Python no sabe evaluar texto en combinación con una operación matemática.

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)
```

2017-07-27

# PLN con Python

└─ Introducción

└─ Operaciones

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)
```

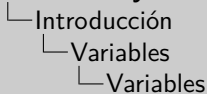
Esto provocará un error.

# 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)
```

# PLN con Python



- 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)
```

- Las variables se "evalúan" como si fueran operaciones.
- En los ejemplos vemos que se utilizan las variables dentro de la función `print()` de la misma manera que las operaciones. Y de la misma manera, se procesan (y se reemplaza por su valor) antes de ser mostradas en pantalla.

# 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.
```



# PLN con Python

## Introducción

### Comentarios

### Comentarios

**IMPORTANTE**

Dejen que su código les hable.  
Y hágalos 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.
```

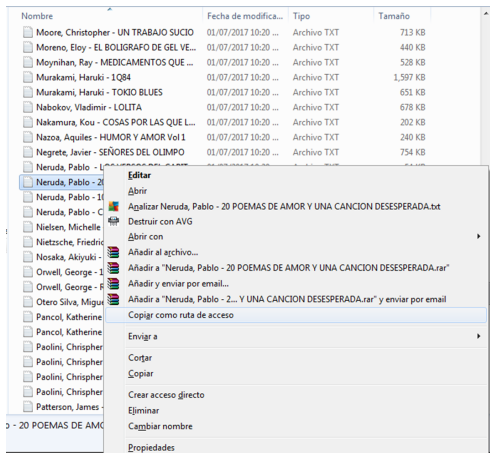
- No me canso de repetir lo importante que es esto.
- Comentar el código es una de los mejores hábitos que se pueden hacer al comenzar a programar.
- Les facilita a recordar lo que hacen las funciones.
- Pero más importante que eso, les ayuda a leer su código, incluso si tiene mucho tiempo que lo hicieron. Entre más saben programar y más complejos son sus programas, pueden plasmar en código ideas mas completas, en el momento las tienen bien presentes, pero cuando se regresa, uno se da cuenta que no recuerda exactamente lo que estaba pensando en ese momento ni para qué hizo las cosas que hizo.
- También para trabajar en equipo sobre un proyecto mas completo, esto es crucial.
- Leer código no comentado es una cosa muy tediosa y difícil. Más aún si es de otra persona.

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.

# PLN con Python

## Archivos

### Escritura de archivos

### Escritura de archivos

#### Escritura de archivos

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

```
archivoabierto=open("C:\\DATA\\mi_archivo_nuevo.txt","w")
archivoabierto.write("Esto se escribe en el archivo\n")
archivoabierto.write("Esto tambien\n")
archivoabierto.write("Mira, puedo escribir \"comillas\"")
archivoabierto.write("Gracias a la diagonal invertida: \\ \"n\"")
archivoabierto.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.

- La diagonal invertida es un símbolo especial, recuerden.
- Este es un ejemplo del por qué, dependiendo de la tecla que siga a la diagonal invertida (en este caso la 'n') tendrá diferente significado.
- \n quiere decir salto de línea, otro ejemplo es \t, que quiere decir tabulador, o la misma \\ que quiere decir diagonal invertida.

# 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()
```

# PLN con Python

## └ Archivos

### └ Ejemplo

Ejemplo

```
c="C:\\Users\\user\\Desktop\\Documentos\\"  
s="P_1PT_290216_73_Acc.txt"  
n="archivo_nuevo.txt"  
s2=open(c+s,"r")  
s2=open(c+n,"w")  
t=s2.read()  
t2=t  
s2.write(t2)  
s2.close()  
s2.close()
```

- Este es un programa de ejemplo que abre dos archivos. Uno para leer y otro para escribir
- Toma el texto del primer archivo y lo escribe en el segundo.
- Es una copia de archivos vaya.

# 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()
```



# PLN con Python

## └ Archivos

### └ Ejemplo

Ejemplo

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IPT_290216_73_8cc.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()
```

- Este es el mismo programa que el anterior.
- Pero quiero que noten cómo es más fácil leer un programa en el que las variables tienen nombres útiles con respecto a para lo que se usan.
- Ambos programas son correctos y hacen lo mismo, pero un programa como este segundo es mucho más recomendable, se vuelve más fácil de leer.
- Y eso aún si agregarle comentarios. Pongan comentarios.

# 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...
```

# PLN con Python

## └ Archivos

### └ 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 sí misma el cierre del archivo.

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documents\\"  
archivo_nombre="P_1PT_200216_T3_Acc.txt"  
  
with open(carpeta_nombre+archivo_nombre,"r") as archivo:  
    texto=archivo.read()  
    print(texto)  
  
# El programa continúa acá...
```

- Notarán que las instrucciones están "invertidas"
- La variable `archivo` se está asignando sin un igual, debido a la instrucción `with`
- Por lo tanto, esa línea de código quiere decir eso, que con (eso quiere decir el `with`) el resultado de la instrucción `open()` asignado a la variable `archivo` se haga lo que se pone a continuación (después de los puntos :)
- Otra forma de verlo, es que la instrucción `with` asigna temporalmente una variable, en este caso, es como si tuviéramos: `archivo = open(...)` pero únicamente válido por el bloque que sigue, es por eso que con esta forma no se necesita cerrar el archivo, se hace solo.

- 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 sí misma el cierre del archivo.

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_1PT_200216_T3_Acc.txt"

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

# El programa continúa acá...
```

- Este ejemplo sirve para mostrar otro asunto incluso más importante.
- El sangrado, mejor conocido como indentación.
- Notarán que inmediatamente después del **with ... as archivo:** sigue un bloque que tiene espacios del lado izquierdo (se recomienda el uso de 4 espacios para ser precisos).
- Python reconoce esas sangrías como una forma de agrupar el código, es la forma de decirle dónde comienza y termina ese **with ... as archivo:**. Y son obligatorios.
- Si se regresa de nuevo a escribir código sin sangría (como el comentario del final) quiere decir que eso ya está fuera del bloque, en este caso, en ese lugar ya no: **archivo = open(...)**.

# Programación

# El 'if'

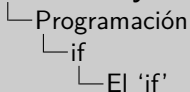
- ▶ El `if` es una instrucción que ejecuta un bloque si una condición se cumple.
- ▶ Es la instrucción más simple de control

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

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!?")
```

# PLN con Python



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_1PT_290216_73_Acc.txt"
palabra="Mexico"

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!")
  
```

- Observen que estamos utilizando la instrucción `in` que busca texto dentro de otro texto, en este caso, estamos buscando la aparición del texto dentro de la variable `palabra` en el texto dentro de la variable `texto`. La instrucción, va a devolver VERDADERO si la palabra aparece en el texto, FALSO si no aparece.
- El `if` va a recibir el resultado, si es VERDADERO se va a ejecutar el bloque de código (identificado por la sangría), de lo contrario, se salta el bloque.
- Para quienes sean mas de números, el segundo `if` revisa una condición lógica, si 2 es mayor que 5, esto también tiene un carácter de verdadero/falso. En el programa deberán notar que ese bloque se salta, el texto "Dos es mayor que cinco!?" nunca aparece.
- Un último detalle en esta lámina. vean como en el bloque de `with` se asigna la variable `texto`, el bloque se cierra, pero más adelante volvemos a usar la variable. Esto esta bien. la condición que se cumple SOLO dentro del bloque de `with` es: **`archivo = open(...)`**.

## 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_290216_73_Acc.txt"
palabra="algo"

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`



# PLN con Python

## └ Programación

### └ if

#### └ if ... else

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="FIFTY_990216_73_acc.txt"
palabra="algo"
```

```
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`

- Pueden cambiar la palabra a la del ejemplo anterior para que vean como el bloque del `else` es ignorado.
- Es importante que noten que el `else` está FUERA del bloque del `if` y además, crea su propio bloque de instrucciones.

## if ... elif ... else

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

```
palabra1="acuerda"
palabra2="ACUERDA"

with open(carpeta_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 :(")
```

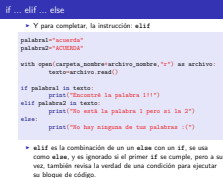
- **elif** es la combinación de un **else** con un **if**, se usa como **else**, y es ignorado si el primer **if** se cumple, pero a su vez, también revisa la verdad de una condición para ejecutar su bloque de código.

# PLN con Python

## └ Programación

### └ if

#### └ if ... elif ... else



- Una pequeña nota para esta diapositiva, observen las dos palabras que se están usando, una está compuesta por minúsculas, la otra por mayúsculas, la segunda se encuentra, la primera no.
- Esto es para aclarar que la instrucción **in** busca coincidencias EXACTAS.
- Y eso también implica que si buscan una palabra que este contenida en otra, **in** la va a encontrar, **in** NO SEPARA palabras.
- Para cosas así, ya veremos mejores herramientas más adelante.

- ▶ Modifiquen el ejemplo anterior para que el programa les indique específicamente cuál palabra fue la que encontró.

# 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)
```

# PLN con Python

## Programación

### Listas

### 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 "+str(longitud_lista))

sum_elemento=sum(semana_laboral)
print("Suma de los elementos "+str(sum_elemento))

semana_laboral.append("Viernes")
print("Semana laboral (append('Viernes')) "+str(semana_laboral))

del semana_laboral[4]
print("Semana laboral (del índice 4) "+str(semana_laboral))

semana_laboral.remove("Viernes")
print("Semana laboral (remove('Viernes')) "+str(semana_laboral))
```

- Noten que la función `len()` regresa el tamaño de una lista, es decir, cuántos elementos contiene, en este caso, la lista tiene 5. Pero RECUERDEN que los índices de la lista comienzan en cero, entonces el último elemento tiene el índice 4. es decir, el valor que dice la función `len()` -1.
- La función `index()` de la lista sirve para obtener el lugar de un elemento dentro de ella misma. Sí utilizan el resultado con los corchetes (como en `semana_laboral[x]`) obtendrán el mismo valor.
- La función `append()` de las listas se usa para agregar valores a la lista (se agregan al final de la lista). Pueden usar nuevamente la función `len()` para ver que la lista crece.



# PLN con Python

## Programación

### Listas

### 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)

sum_elemento=semana_laboral.index("Viernes")
print("Lugar del Viernes "+,sum_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)
```

- El comando `del` se usa para eliminar un elemento de la lista. Tengan cuidado, si en lugar de colocar el elemento (con índice) y colocan solo el nombre de la lista, se borrará la lista completa.
- La función `remove()` de las listas, se usa para eliminar un elemento de la lista según su valor (a diferencia de `del` que usa el índice). Nota: se va a eliminar solo la primera aparición, si tienen el valor repetido, no se borrarán todos.

# 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="P_IFT_290216_73_Acc.txt"

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

print(lineas_lista)
```

# Bucles

'for'

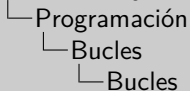
- ▶ Los bucles son los procesos iterativos.
- ▶ La misma acción se hace una y otra vez para un conjunto de datos, por lo general para todos los elementos de una lista.
- ▶ Por ejemplo, podemos hacer que para cada línea de un archivo Python la escriba en la pantalla:

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

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

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

# PLN con Python



- 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\\"
archivo_nombre="P_197_290215_73_Acc.txt"

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

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

- Presentamos el 'for'. La instrucción **for** lee una lista, y para cada uno de sus elementos ejecuta un bloque de instrucciones.
- En nuestro ejemplo, la variable **lineas** es una lista que contiene las líneas del texto.
- Con la línea: **for linea in lineas** : lo que estamos haciendo es decirle a la instrucción **for** que recorra todos los elementos de la lista **lineas**. El elemento que esté leyendo en ese momento, lo va a almacenar en la variable **linea** para que se pueda manejar dentro del bloque de instrucciones. Cuando termine el bloque de instrucciones, pasará al siguiente elemento de la lista y lo asignará a la variable **linea** y así sucesivamente.

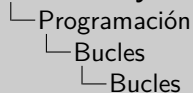
# Bucles

'for'

- Veamos un ejemplo un poco mas complejo:

```
01: carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
02: archivo_nombre="P_IFT_290216_73_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")
```

# PLN con Python



▀ Veamos un ejemplo un poco mas complejo:

```
01: carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
02: archivo_nombre="P_1PT_350216_73_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")
```

- ¿Qué está ocurriendo aquí? Este programa muestra el documento en pantalla, pero precede a cada línea con texto que indica el número de línea que es. Esta vez les puse las líneas del ejemplo enumeradas para que sea mas sencillo de seguir.
- En las líneas 1 y 2 se están definiendo la ruta y el nombre del archivo con el que vamos a trabajar, ya hemos visto bastante esto. En las líneas 4 y 5 abrimos y leemos el archivo, también seguramente tenemos esto ya dominado. A partir de este momento, la variable `lineas` contiene una lista con todas las líneas del archivo.
- En la línea 7 se define la variable `num_linea`. Su objetivo es indicar el número de la línea del texto que se está leyendo, comenzaremos en la línea 1, por eso su valor inicial es 1. Ese valor cambiará dentro del bucle.

▀ Veamos un ejemplo un poco mas complejo:

```
01: carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
02: archivo_nombre="P_197_390216_73_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")
```

- De las líneas 8 a 10 se encuentra nuestro bucle, en la línea 8 tenemos la instrucción **for**, que nos indica que se va a recorrer la lista **lineas** y que dentro del bloque de instrucciones, el elemento que se esté manejando se va a llamar **linea**.
- En la línea 9 tenemos la función **print()**, que va a mostrar en pantalla el texto: "LINEA", seguido de la variable **num\_linea** (recordemos que tiene el valor 1, es decir va a mostrar: "LINEA 1", seguido de dos puntos, seguido del contenido de la variable **linea** (es decir, del texto que tenga la línea que se esté procesando en ese paso del bucle).

# PLN con Python

## Programación

### Bucles

### Bucles

► Veamos un ejemplo un poco mas complejo:

```
01: carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
02: archivo_nombre="P_197_390216_73_Acc.txt"
03:
04: with open(carpeta_nombre+archivo_nombre, "r") as archivo:
05:     lines_lista=archivo.readlines()
06:
07: num_linea=1
08: for linea in lines_lista:
09:     print("LINEA", num_linea, ":", linea)
10:     num_linea=num_linea+1
11:
12: print("FIN DE ARCHIVO")
```

- En la línea 10 se le asigna a la variable `num_linea` el valor que tenía más uno. Es decir, se incrementa `num_linea`, eso quiere decir que para el siguiente paso del bucle, en lugar de ser 1, va a ser 2! Además, en el siguiente paso del bucle se volverá a incrementar, por lo que para el tercer paso será 3 y así sucesivamente.
- Por último, en la línea 12 se muestra en pantalla el texto: "FIN DE ARCHIVO". Este texto se encuentra fuera del bucle, y el programa no llegará a ese punto hasta haber terminado los ciclos. Ese mensaje lo veremos al final de todas las líneas del texto.



# Bucles

'for'

- Los bloques de código se pueden anidar.

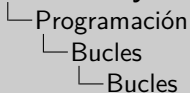
```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_IFT_290216_73_Acc.txt"

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

num_linea=1
for linea in lineas_lista:
    if "." in linea:
        print("La línea",num_linea,"contiene al menos un punto.")
        num_linea=num_linea+1

print("\nLas otras líneas no tienen puntos")
```

# PLN con Python



► Los bloques de código se pueden anidar.

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_DPT_290216_73_acc.txt"

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

num_lineas=1
for linea in lineas_lista:
    if "." in linea:
        print("La linea",num_lineas,"contiene al menos un punto.")
    num_lineas=num_lineas+1

print("¡¡Las otras líneas no tienen puntos")
```

- ¿Si recuerdan la instrucción **in**? Es una instrucción que verifica si un texto contiene otro, en este caso, si hay un "." dentro de **linea**.
- La instrucción **in** regresa un valor de verdadero/falso por lo que podemos usar la instrucción **if** para hacer una serie de instrucciones si es que efectivamente hay un punto dentro de la línea.
- La instrucción **if** tiene su propio bloque de instrucciones, pero todo está dentro del bloque de instrucciones del **for** y no hay ningún problema, es válido.
- Noten, sin embargo, que la sangría si se distingue, todo lo que está dentro del bloque de la instrucción **for** está a un nivel, incluyendo al la instrucción **if**, pero todo lo que está dentro del bloque de la instrucción **if** está un nivel más abajo, por lo tanto tiene aún más sangría.

# 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 punto, indique que esa línea no tiene punto.
- ▶ Muestre cuántas líneas tienen punto 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")
```

# 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)
```

```
carpetas_nombre = ("C:\\Users\\[user]\\Desktop\\Documents\\")
archivo_nombre = "TP_TFT_29026f6_73_acc.txt"

with open(carpetas_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)
```

- Analicemos este programa.
- La primera parte consiste en ubicar nuestro archivo, abrirlo, leerlo y almacenar el texto en la variable `texto`, hasta aquí hemos visto esto repetidas veces.
- Luego viene una nueva variable: `simbolos`. Que es una lista con los símbolos que queremos quitar de nuestras palabras separadas.
- Mas adelante viene un bucle sobre cada uno de esos símbolos.
- Dentro de ese bucle, usamos una nueva función, la función `replace()` de los textos (las variables de tipo texto las tienen) y es un equivalente a un "buscar y reemplazar", si pueden notar, tiene de entrada dos argumentos, el primero es lo que debe buscar, el segundo, con lo que debe reemplazar.

# PLN con Python

- Programación
  - Palabras
    - Tokenización

```
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="1_27_200218_78_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)
```

- Nuestra función `replace()` cambia todos los símbolos para que estén rodeados de espacios.
- De esta manera, la función `split()` los separará como si se trataran de otras palabras.
- La última parte del código ya la vimos en el ejemplo anterior, se separa (con la función `split()`) el texto usando todos espacios como puntos de corte dentro de una lista. Se ordena la lista en orden alfabético, y se muestra en pantalla elemento por elemento.

# 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))
```



► Comencemos ahora con los símbolos especiales de las expresiones regulares.

► Veamos el punto '.': El punto coincide con un carácter, el que sea.

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

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

- En la diapositiva se muestran dos ejemplos de código, uno usando la función `search()` y el otro usando la función `finditer`.
- En el primer caso, verán que se va mostrar la primera letra del texto, el punto va a coincidir con esa letra, cualquiera que sea, y listo, la regresa y termina.
- En el segundo caso, verán que se mostrará todo el texto, letra por letra, esto es debido a que el punto va a ir coincidiendo con cada una de las letras del texto, una tras otra y las va a regresar como resultado. Cuando se hace el bucle sobre todos los resultados, se mostrarán uno a uno, es decir, letra a letra, espacio a espacio, símbolo a símbolo, el punto coincide con todo EXCEPTO el salto de línea (`\n`)

# 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))
```

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

```
expresion_regular=re.compile(r".*")
resultado_búsqueda=expresion_regular.search(texto)
print(resultado_búsqueda.group(0))

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

- En el programa de ejemplo, simplemente estamos añadiendo un asterisco a la expresión regular, con lo que ahora en lugar de estar buscando "cualquier caracter", ahora estamos buscando "cualquier caracter, cualquier cantidad de veces". Lo que es casi equivalente a buscar cualquier cosa.
- Para el segundo ejemplo, de hecho, verán que se muestra en pantalla el texto completo.
- Para el primero, sin embargo, verán que solo aparece el primer párrafo, o mejor dicho, la primer LINEA del texto, esto es debido a que el punto NO COINCIDE con salto de línea, es lo único con lo que no coincide, por lo tanto, la expresión (que solo regresa la primera coincidencia en el primer ejemplo) toma todo hasta el primer salto de línea y termina.

# 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))
```

▸ 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))
```

- Este programa los puede confundir un poco, verán que en primer programa, no aparece NADA, mientras que en segundo programa, aparecen muchísimas líneas vacías con alguna que otra 'I' por allí, e incluso varias repeticiones de 'I's.
- ¿Qué es lo que está pasando? El asterisco sirve para encontrar cualquier número de repeticiones, de 'I' en este caso, pero eso incluye 0 repeticiones.
- Para el primer ejemplo, eso quiere decir que si al inicio no encuentra una 'I', no importa, porque pueden ser 0, así que ya terminó.

▸ 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_buqueda=expresion_regular.search(texto)
print(resultado_buqueda.group(0))

expresion_regular=re.compile(r"i+")
resultados_buqueda=expresion_regular.finditer(texto)

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

- Alguien podría notar que entonces, eso también aplicaría para el ejemplo del punto con asterisco, que siempre que se use un asterisco el resultado podría estar vacío. Eso es parcialmente cierto, pero las expresiones regulares SIEMPRE buscan el resultado más largo que puedan obtener.
- En el segundo programa, aparecen muchas líneas vacías (replicando el comportamiento del primer programa para cada letra diferente de 'i'). Pero cuando si aparece una 'i', el programa va a regresar TODAS las que estén juntas, no se limitará solo a encontrarlas, es por eso que verán algunas apariciones de varias 'i's juntas.

# 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))
```

- 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_búsqueda=expresion_regular.search(texto)
print(resultado_búsqueda.group(0))

expresion_regular=re.compile(r"i+")
resultados_búsqueda=expresion_regular.finditer(texto)
for resultado in resultados_búsqueda:
    print(resultado.group(0))
```

- Seguramente este programa tendrá un resultado mucho más parecido al que esperaban.
- El símbolo de suma garantiza que al menos una letra de las que se busca aparecerá (si es que se encuentra en el texto) y además, tiene el mismo comportamiento de buscar tantas repeticiones como sean posibles.
- Las expresiones regulares:  $I^+$  y  $II^*$  son equivalentes.



# 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))
```

- Pasamos al siguiente símbolo especial: `'''`
- Se utiliza para indicar el inicio del texto.

```
expresion_regular=re.compile(r"''")
resultado_buqueda=expresion_regular.search(texto)
print(resultado_buqueda.group(0))

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

- Nota, ese símbolo por lo regular está a la derecha de la 'Ñ' en el teclado, para sacarlo, tienen que mantener oprimido ALT\_GR y dos veces esa tecla.
- En el resultado del programa, notarán que se tiene la misma salida para ambos programas, esto es porque en ambos casos solo se tiene un inicio del texto.
- Si leen, por ejemplo, el texto por líneas, y hacen un recorrido de cada línea buscando ese patrón, obtendrán como resultado la primera letra de cada línea.

- ▶ 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))
```

# PLN con Python

## └ Expresiones Regulares

### └ Expresiones regulares

#### 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.findall(texto)
resultados_cortos=expresion_corta.findall(texto)

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

- Una aclaración importante, este programa busca texto que se encuentre entre paréntesis, pero observen que los paréntesis tienen una diagonal invertida (\) antes de ellos. Esto es porque el paréntesis también es un símbolo especial, lo veremos más adelante.
- Del resultado del ejemplo, podemos ver la diferencia de utilizar el signo de interrogación o no junto con el asterisco, en el primer caso, se toma el primer paréntesis que aparece en el texto y no se "cierra" hasta que se encuentra el último cierre de paréntesis. En el segundo ejemplo, se busca el primer paréntesis para el cierre de cada uno de los que se "abren".
- Les puse un `print()` extra entre los dos `for`. Eso es solo para poner un espacio blanco y que se vean separados los dos resultados para mayor claridad.



# 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))
```

- 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) articulo?')  
resultados_búsqueda=expresion_regular.findall(texto)  
  
for resultado in resultados_búsqueda:  
    print(resultado.group(0))
```

- Este programa tiene un comportamiento muy similar al anterior, con la diferencia de que no regresa uno de los resultados del anterior, uno en la que la palabra "artículos" aparece sin ninguna de las dos opciones que le estamos dando. Dado que en el primer ejemplo ambas son opcionales, captura la aparición. En este ejemplo, estamos forzando a que al menos una de las opciones aparezca, o no se recupera el texto.
- Pueden agregar el símbolo de interrogación a la disyunción (lo que está entre paréntesis) y lograrán el mismo efecto anterior.

# 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))
```

- 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"[^a-z][^0-9]([a-z])+")
resultado_buqueda=expresion_regular.findall(texto)

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

- De estos dos últimos ejemplos, podemos notar que el recorrido de la 'a' a la 'z' no incluye letras con acentos.
- Además de los recorridos de abecedario, también se pueden hacer recorridos de dígitos numéricos.

# 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))
```



- ▶ 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+)?")
resultados_buqueda=expresion_regular.finditer(texto)

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

- La expresión regular de este último ejemplo será capaz de encontrar todos los números de un texto, sea que tengan punto decimal o no, y ya sea que estén separados por comas o no.
- Noten que para representar el punto, y que signifique punto en lugar de "cualquier caracter" le ponemos una diagonal invertida (`\`) antes de. La diagonal invertida le da significado especial a ciertas letras (como a la 'd') pero también le quita su significado especial a las que la tienen.

# 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))
```

- 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_búsqueda=expresion_regular.finditer(texto)

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

- En el ejemplo se buscan por todas los elementos diferentes de espacios que estén juntos.
- Si se fijan, el resultado es equivalente a cuando separamos todo el texto usando los espacios como divisiones.

# 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))
```

- Un grupo muy importante también: \w
- Este conjunto incluye todos los caracteres que se esperaba 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_búsqueda=expresion_regular.finditer(texto)

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

- Podrán notar que el resultado del ejemplo también divide el texto, solo que utiliza tanto espacios como cualquier otro signo de puntuación, y básicamente cualquier cosa que no se encuentre dentro del conjunto usado.

- ▶ 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"

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".

- Estas consideraciones no cubren las menciones de TODOS los artículos, hay un par que se encuentran más revueltos en el texto, pero por ahora solo tomen en cuenta estas tres consideraciones.



# 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")
```

Por último hablemos de grupos y referencias.

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

```
expresion_regularr=re.compile(r"([a-z]+)+")
resultados_búsqueda=expresion_regularr.finditer(texto)

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

Así que, qué hace este programa?

- Busca una palabra que se encuentra entre paréntesis (observen que los paréntesis están precedidos por una diagonal invertida (\) para que pierdan su significado especial).
- Noten que dentro de los paréntesis con diagonal invertida (que son los que coincidirán con paréntesis del texto) hay otros paréntesis SIN diagonal invertida. Éstos están formando un grupo, para el cual estamos buscando una palabra que comience con una letra mayúscula.
- Ya fuera del paréntesis, busquemos cualquier cosa (. \* es cualquier caracter cualquier cantidad de veces, así que es lo que sea) hasta que vuelva a aparecer lo que encontró dentro de los paréntesis (el grupo tiene el nombre de \1).
- Por último se muestra en pantalla la coincidencia total, y además mostramos lo que coincidió entre paréntesis (los números también coinciden con los grupos del resultado, excepto el 0 ya que no hay \0).

# 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)
```

- 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_regul= re.compile(r"([a-zA-Z])"+"")
texto_nuevo=expresion_regul.sub(r" \1", texto)
print(texto_nuevo)
```

- La función `sub()` tiene dos parametros (dos datos de entrada separados por una coma) el primero es la expresión con la que se quiere reemplazar la expresión regular definida (definida con la función `compile()` esto ya lo hemos visto). Y el segundo es el texto sobre el que se quiere hacer la sustitución.
- Para poder utilizar el grupo `\1` en la sustitución, tenemos que usar paréntesis, en esta ocasión, alrededor de toda la expresión.
- Este programa lo que está haciendo es buscar signos de puntuación (en realidad estamos buscando cosas diferentes a contenidos de palabras o espacios, y con eso prácticamente solo quedan símbolos) a los que no les sigan ni letras mayúsculas ni dígitos (un símbolo seguido de una letra mayúscula o un dígito puede significar siglas, abreviaturas o números).

- 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"([a-zA-Z])"+"")
texto_nuevo=expresion_regular.sub(r" ",texto)
print(texto_nuevo)
```

- Una vez que encuentra esos signos (presuntamente que no están ni entre letras ni entre números) les agrega un espacio del lado izquierdo (para separarlo del texto en el que está).
- Este es el primer paso para separar bien los signos que se tienen que separar con espacios. Si observan el resultado, verán que ahora los signos están correctamente separados cuando se encontraban a la derecha de las palabras, pero aún no cuando se encuentran a la izquierda (como los paréntesis sobre todo).
- La función `sub()` regresa el texto reemplazado, este texto se puede guardar en una variable nueva o incluso sustituyendo el texto antiguo para seguir procesándose más adelante.

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

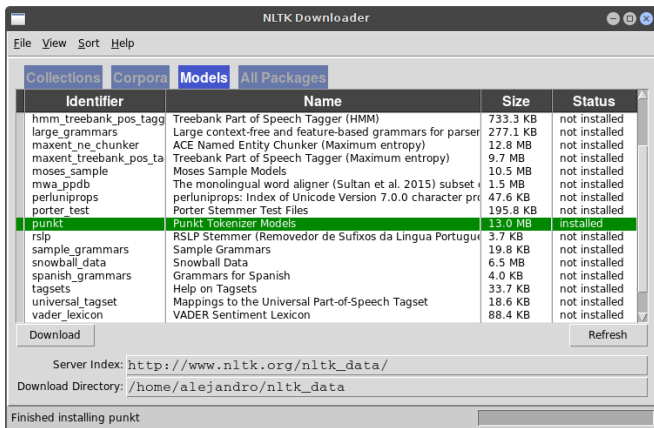
- ▶ NLTK (Natural language tool kit) es un módulo de Python que contiene herramientas para el manejo del lenguaje natural.
- ▶ Para usarlas, al igual que con los otros módulos, se tiene que importar.

```
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)
```

# PLN con Python

└─NLTK

└─NLTK

NLTK

Tokenizador

► Y ahora que ya descargamos el tokenizador, podemos cerrar esta ventana y usarlo. De regreso en Python.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_1P7_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)
```

- Ya sabemos como abrir y leer archivos.
- El tokenizador de NLTK (la función `word_tokenize()` usa el texto leído y regresa una lista con los tokens que encuentra.
- Como es una lista, podemos hacer un bucle para recorrerla y mostrar el resultado.
- Notarán algo, estamos dando un segundo parámetro a `word_tokenize()` para indicar que nuestro texto está en español (de allí que el parámetro es: "spanish").
- El tokenizador tiene varios modelos, adaptados para diferentes idiomas, si tienen la curiosidad y el tiempo, pueden entrar a la carpeta en la que se descargó en tokenizador para que vean los idiomas que sabe manejar.

► Y ahora que ya descargamos el tokenizador, podemos cerrar esta ventana y usarlo. De regreso en Python.

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_1P7_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)
```

- En cuanto al resultado, pueden comparar los tokens que se obtienen con este método contra los de nuestro pequeño tokenizador de expresiones regulares.
- Verán que, por ejemplo, este detecta mejor abreviaciones como "S.A." ya que el nuestro separa el último punto. Pero este no separa los punto y guión (.-) de las palabras que tiene junto, y el nuestro sí.
- Las reglas y los procesos que se usan en cada herramienta varían, es importante revisar los resultados que se tienen y escoger la mejor herramienta para cada tarea.



# 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)
```

- ▶ 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)
```

- ¿Recuerdan la función `len()` de las listas? Nos proporciona el tamaño de la lista.
- Si tomamos en cuenta que la lista de tokens contiene (separados) cada uno de los elementos que tiene el texto, bastará con ver qué longitud tiene para saber cuántos elementos tiene el texto.
- Esto va a incluir símbolos y puntuación, tengan eso en mente.

# 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
```

# PLN con Python

## └ NLTK

## └ Funciones

- 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_con_junto = set(tokens)
    palabras_totales = len(tokens)
    palabras_diferentes = len(tokens_con_junto)
    riqueza_lexica = palabras_diferentes / palabras_totales
    return riqueza_lexica
```

- Las funciones se crean con la instrucción especial `def` seguido del nombre de la función y los paréntesis (toda función lleva paréntesis) dentro de los que se pueden usar parámetros.
- Los parámetros son básicamente variables que se asignan al momento de llamar la función. En el caso del ejemplo, el parámetro `tokens` se puede usar dentro de la función como una variable. Su valor estará dado por lo que se ponga entre los paréntesis en el momento de usar la función (más adelante en el programa).
- Las variables dentro de una función son válidas **UNICAMENTE DENTRO DE LA FUNCIÓN**. Eso quiere decir que si más adelante en el programa, fuera de la función, se intenta usar una de las variables definidas dentro de la función no va a funcionar.
- La instrucción `return` es una instrucción especial de las funciones, llegando a ese punto, la función termina y regresa el valor especificado. Esto quiere decir que ese resultado se podrá asignar a una variable más adelante.

# 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)
```

```
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_377_590216_73_acc.txt"

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

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

- Aquí se muestra un ejemplo más completo.
- Noten que las funciones se definen pronto dentro del archivo de Python, esto facilita a que no se pierdan las funciones entre el resto del código, y para que estén disponibles desde el principio. Las funciones SOLO están disponibles DESPUÉS de que fueron definidas.
- Como pueden ver, la función se usa más adelante, después de que fue creada, y los nombres de las variables NO AFECTAN a la función. Es decir, la función fue definida con un parámetro de entrada `tokens`, pero eso es solo para la función, fuera de ella, se puede usar el nombre independientemente, es más, como se ve se puede usar el nombre `tokens` para guardar la información que más tarde se le pasa a la función aunque tenga el mismo nombre del parámetro.



```
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_377_590216_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)
```

- Otro detalle que ilustra el ejemplo está en los nombres de las funciones. Observen como la función y la variable a la que se asigna el resultado de la función, se llaman igual. Esto es correcto, no pasa nada, ya que la diferencia está en que parte del nombre de la función son los paréntesis, una función se diferenciará de una variable del mismo nombre siempre por los paréntesis.
- Además de eso, aquí se puede ver cómo es que el resultado de la función se asigna a una variable, esto es gracias a la instrucción **return** que finaliza la función, el valor que estaba en la variable **riqueza\_lexica** de la función se devuelve luego de ser llamada. Y, en el caso del ejemplo, se guarda en OTRA variable que se llama **riqueza\_lexica**, todo eso ya FUERA de la función.

- ▶ 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)
```

- 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_1PT_200216_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)
```

- Las variables tipo `Text` de NLTK muchas veces también se comportan como listas.
- Por ejemplo, pueden usar su función `count()` para ver las repeticiones de una palabra, y pueden también usar la función `len()` para ver la longitud total de palabras o la función `set()` para convertirla en un conjunto.

# 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")
```



# PLN con Python

└─NLTK

└─NLTK

## NLTK

### Concordance

- 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_DFT_000016_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("articulo")
```

- Es importante notar, que las concordancias de NLTK buscan según los tokens SIN importar mayúsculas y minúsculas, pero sí la coincidencia completa de la palabra.
- Esto quiere decir, que el ejemplo sería capaz de encontrar la palabra "Artículo" si la hubiera, pero no "artículos" aunque la haya.
- Pueden probar buscando la palabra "ley" para ver que las mayúsculas y minúsculas se buscan por igual.
- Otro detalle, la función `concordance()` es similar a la función `print()` en cuanto a que su propósito es mostrar ese resultado en pantalla, NO funciona para mandar el resultado a una variable.

# 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))
```

# PLN con Python

└ NLTK

└ RE

## RE Concordancia

- 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

# Aquí va la lectura del archivo, por ahora la estoy omitiendo
expresion=re.compile(r'^(.{,30})[Aa]rticulo?' + '{,30}$')
resultados_busqueda=expresion.findall(texto)

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

- Los corchetes sirven para especificar un número concreto de repeticiones del símbolo que les precede. Esto quiere decir que si tenemos, por ejemplo: `.{3}` Quiere decir que se buscan 3 letras, las que sean.
- Además de eso, el corchete puede usarse para rangos delimitados de repeticiones. Esto quiere decir que podemos buscar, por ejemplo, entre 2 y 6 caracteres: `.{2,6}`.
- Si se omite uno de esos dos números, quiere decir que no hay límite, como en el ejemplo, que se busca de cero (por eso no hay número, es igual si se pone 0) hasta 30 caracteres en ambos lados de la palabra.
- Con las expresiones regulares se tienen que tomar en cuenta más cosas para lograr una buena concordancia, pero de igual manera, se pueden lograr mas cosas, en este ejemplo, estamos incluyendo, por ejemplo, que la palabra pueda aparecer al inicio de la línea, que pueda aparecer con mayúscula en la primera letra, o que pueda contener una s al final. Las primeras dos funciones las cumplen las concordancias de NLTK, la tercera no. Una vez más, depende del objetivo, la herramienta que se debe usar.

# 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)
```

- 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)
```

- Una pequeña nota, observen que a diferencia de las otras funciones que hemos visto, aquí SI importa el uso de mayúsculas.
- Para esta función, es necesario que tengan instalados los paquetes de python `numpy` y `matplotlib`. Si instalaron Anaconda, seguramente ya cuentan con ambos.



# 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)
```

# PLN con Python

└ NLTK

└ NLTK

NLTK

Distribución de frecuencias

```
import nltk
carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_DFT_200216_73_Acc.txt"

with open(carpeta_nombre+archivo_nombre+".txt") 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)
```

- Hemos visto cómo hacer conteos de palabras totales y de palabras únicas.
- Ahora veremos cómo hacer conteos para todas las palabras de un texto. Y para lograrlo podemos usar la función `FreqDist()` de NLTK.
- Como ven, usamos la función sobre nuestro `Text` para obtener una distribución.
- La distribución tiene su propia función para mostrar la información. La función `most_common()` que devuelve una lista con cada palabra y su frecuencia. Tal como se puede ver al mostrar la lista que se obtiene con la función `print()`.
- La función `most_common()`, además, puede recibir dentro de los paréntesis un parámetro numérico, el cual indica la cantidad de cuántas palabras queremos que se obtengan en la lista de salida (las más comunes, claro).

- ▶ 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])
```

# PLN con Python

└ NLTK

└ Diccionarios

## 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])
```

- Para definir un diccionario, se usan llaves (`{}`) en lugar de corchetes (`[]`). Los elementos TAMBIÉN se separan por comas, además, el nombre del dato y el contenido del dato, se separan con dos puntos.
- Las funciones que vimos que operan sobre listas (como `del()` y `len()`) también funcionan con diccionarios.

- ▶ 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)
```

# PLN con Python

└ NLTK

└ NLTK

## 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(cumulative=True)
```

- Le agregué unas líneas al programa para que se muestre el número total de palabras y el número de palabras diferentes. Son cosas que ya hemos hecho antes, confío en que lo recuerdan y será claro al verlo.
- Por cierto, una nota importante. Recuerden que los programas de computadora ejecutan UNA instrucción a la vez, por lo que tanto en este ejemplo como en el anterior, para que se muestre la segunda gráfica tiene que terminar de mostrar la primera. Es decir, tienen que cerrarla.

- ▶ 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

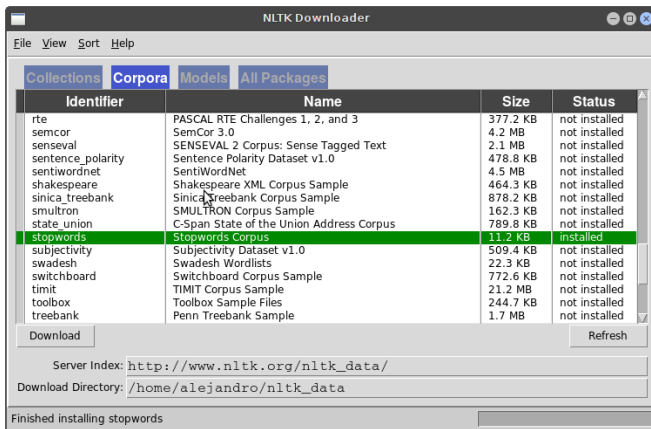
## Palabras funcionales

- ▶ 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))
```

# PLN con Python

└ NLTK

└ NLTK

NLTK

Palabras funcionales

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_097_200216_73_acc.txt"

with open(carpeta_nombre+archivo_nombre+".") 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))
```

- ¿Qué estamos haciendo aquí?
- Estamos filtrando las palabras funcionales y las quitamos del texto.
- Tenemos nuestra lista de tokens, dentro de la variable **tokens**.
- Inmediatamente después, creamos una nueva variable: **tokens\_limpios**. Que como pueden ver, es una lista vacía! El plan, es llenarla a continuación.
- Y para eso justamente está ese **for**. Usamos un bucle que recorre todos los tokens de nuestro texto y verificamos si son palabras funcionales o no.
- Para ver si son palabras funcionales, usamos la instrucción **not in** que como es de esperar, hace lo contrario a la instrucción **in** (la recuerdan?).

# PLN con Python

└ NLTK

└ NLTK

```

NLTK
Palabras funcionales

import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_027_200216_73_acc.txt"

with open(carpeta_nombre+archivo_nombre+".") 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))

```

- En resumen, ese `if`, ve si el token que se está revisando está dentro de la lista de palabras funcionales. Si NO lo está (por eso el `not in`) entonces podemos decir que ese token no es una palabra funcional. Por lo tanto queremos que esté dentro de nuestra lista limpia de palabras funcionales.
- Una vez que la condición se cumple y que vemos que el token NO es una palabra funcional, lo agregamos a la lista que habíamos creado (a la lista de `tokens_limpios`) con la función `append()`. Esa función lo que hace es agregar el valor que le damos al final de la lista.
- Por último, mostramos en pantalla las longitudes de ambas listas de tokens, para que vean la diferencia de tamaño entre la que tiene palabras funcionales y la que no.

- ▶ 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)
```

# PLN con Python

└ NLTK

└ NLTK

## NLTK

### Palabras funcionales

```
import nltk

carpeta_nombre="C:\\Users\\user\\Desktop\\Documentos\\"
archivo_nombre="P_DPT_200216_73_acc.txt"

with open(carpeta_nombre+archivo_nombre+".") 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(400)
```

- El cambio está en el bucle en el que se llena la lista `tokens_limpios`. Agregamos un `if` dentro del que ya había.
- Esto quiere decir que no bastará ahora con que el token no sea una palabra funcional, si no que ahora, se busca también que: `len(token) > 1`. Y esto quiere decir que el token tenga MAS de una letra.
- La función `len()` la habíamos usado para ver el tamaño de una lista, pero también la podemos usar para ver cuántas letras tiene una palabra, el texto en Python se comporta como una lista de letras.
- En este nuevo programa, si se cumple que el token NO es una palabra funcional Y que tiene MÁS de 1 letra, entonces se agrega a la lista de tokens limpios.
- Esto va a eliminar los símbolos, aunque también elimina otras cosas, hay que tener eso en cuenta.

- ▶ 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=nltk.Text(tokens_limpios)
distribucion_limpia=nltk.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.

NLTK

Colocaciones

- 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.

- La función `collocations()` de NLTK también es una de esas funciones que escribe por si misma en la pantalla, no es necesario asignarla a una variable.

- ▶ 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)
```

# PLN con Python

└─NLTK

└─NLTK

## NLTK

Esquema PoS

- 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)
```

- Lo mejor es importar el etiquetador externo de nltk como se muestra para no tener que usar toda la ruta.



- 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)
```

# PLN con Python

└ NLTK

└ NLTK

## NLTK

Etiquetado PoS

- 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)
```

- Las etiquetas se regresan en una lista, cada elemento tiene el token junto con su etiqueta.
- Notarán que cada pareja (token-etiqueta) dentro de la lista se encuentra entre paréntesis, esto en realidad quiere decir que también están formando una lista.
- Ok, no una lista, sino una tupla, la única diferencia entre ambas es que estas tuplas NO las pueden modificar, digamos que son de solo lectura. Pero a fin de cuentas, pueden ingresar a sus datos por índices igual que lo harían con una lista.
- Así que, si usan por ejemplo `etiqueta[0]` obtendrán el token únicamente, si usan `etiqueta[1]` obtendrán solo la 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)
<b>Adjectives</b>		
ao0000	Adjective (ordinal)	<i>primera, segundo, últimos</i>
aq0000	Adjective (descriptive)	<i>populares, elegido, emocionada, andaluz</i>
<b>Conjunctions</b>		
cc	Conjunction (coordinating)	<i>y, o, pero</i>
cs	Conjunction (subordinating)	<i>que, como, mientras</i>
<b>Determiners</b>		
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>