Claves de Cristeros - Python

July 6, 2021

1 Claves en Cristeros

En el grupo de Cristeros se utilizan distintas claves como medios de comunicación especiales, ya sea como actividades del aspecto de destreza en forma de juegos o competencias, o para transmitir un mensaje de manera discreta entre miembros del una comarca o entre amigos. Las claves por lo general consisten de un sistema por el cual se toma una palabra o frase como entrada, también conocido como texto plano, y por medio de las reglas definidas por dicho sistema se cifra la entrada y se obtiene una salida, distinta por lo general a la entrada, también conocido como texto cifrado. Para fines de este documento se utilizarán los términos entrada y salida, por fines prácticos. Ahora bien, para utilizar este tipo de claves se necesita ya sea conocer el sistema de memoria o una tabla con el sistema impreso o digital que se tome como guía para cifrar y descifrar los mensajes. Existen técnicas y consejos para agilizar el uso de las mismas bajo dicha metodología, sin embargo en esta investigación nos enfocamos en una vía alternativa a la manual: el uso del poder computacional. # Python En este caso se utilizará el lenguaje de programación de Python en su versión 3.8 desde la interfaz de JupyterLab. Para hacer uso de los códigos que se presentarán más adelante se requieren como dependencias los paquetes de **numpy**, para operaciones numéricas, y **pydub**, para manipulación y reproducción de audio. Se escogió Python por su universalidad y la manera en la que maneja las letras y palabras. ## Strings En Python existen diferentes tipos de datos, entre ellos los conocidos como **strings**, que en términos simples consisten en líneas de texto. Para especificar que se desea utilizar un objeto de tipo string se pone el texto entre apóstrofes (' ') o entre comillas ("“). Por ejemplo, si se quisiera crear una variable llamada **word** (del inglés de palabra) que contenga la palabra”Cristeros", se haría de cualquiera de las dos siguientes maneras:

[1]:
```
word = 'Cristeros'
word = "Cristeros"
```

Por lo general se prefiere la primera por ser más compacta. Ahora si queremos ver el valor de la variable word desplegado utilizamos el comando “print”:

[2]:
```
print(word)
```

```
Cristeros
```

1.0.1 Concatenación

Ahora supongamos que quisiéramos agregar después de esto la frase “dar la vida por Cristo”, con una coma en medio. Podemos utilizar la operación de concatenación para juntar tres diferentes

strings en una sola. Concatenamos el valor actual de la variable word con una coma seguida de la frase deseada y actualizamos el valor de la variable word.

[3]:
```
word = word + ', '+'dar la vida por Cristo'
print(word)
```

```
Cristeros, dar la vida por Cristo
```

También pudimos haber utilizdo el operador "+=":

[4]:
```
word += ', '+'dar la vida por Cristo'
print(word)
```

```
Cristeros, dar la vida por Cristo, dar la vida por Cristo
```

1.0.2 Carácteres

Cada carácter de un string es un elemento por separado junto en un solo objeto. Podemos acceder cada elemento del string con índices. Para esto se utiliza **string[índice]**, donde **índice** es un número que indica el número de elemento del carácter. Cabe mencionar que en Python la enumeración de los elementos comienza desde el 0, por lo que el primer elemento es el 0, el segundo es el 1, el tercero es el 2, y así sucesivamente. Si queremos llamar la primera letra de la palabra cristeros hacemos lo siguiente:

[5]:
```
word = 'cristeros'
word[0]
```

[5]:
```
'c'
```

1.1 Funciones

Las funciones son una líneas de código que toman un valor como entrada y nos arrojan una salida. Existen también las rutinas, que con como las funciones pero arrojan varias salidas, pero para fines de este documento se hablará únicamente de funciones, independientemente del número de salidas. En este caso particular, buscaremos diseñar funciones que tomen como entrada un texto plano y nos devuelvan el texto cifrado en la clave que necesitemos. El límite de lo que puede hacer una función es el límite de la mente misma. Se puede empezar muy básicamente creando una función que sume dos números. Primero se utiliza el comando "def" para especificar que se planea definir una función. Después escribimos el nombre de la función y entre paréntesis las entradas que tendrá la función. Si queremos que la función nos de la suma de dos números, tendríamos dos entradas, que podemos llamar a y b:

[6]:
```
def suma(a,b):
    return a+b
```

Después de definir el nombre de la función y las entradas se colocan dos puntos (:) y se utiliza el comando "return" para establecer la salida de la función. En este caso queremos que la función nos regrese la suma de los dos números a y b. Probemos con los números 3 y 4. Para utilizar

una función (llamar una función) simplemnte se escribe el nombre de la función seguido de las entradas dentro de paréntesis.

[7]:
```
suma(3,4)
```

[7]:
```
7
```

Podemos hacer también de la misma manera una función llamada "unir" que junte dos líneas de texto (strings):

[8]:
```
def unir(a,b):
    return a+b
```

Como ejemplo utilizamos "Cristeros" y "SJB":

[9]:
```
unir('Cristeros','SJB')
```

[9]:
```
'CristerosSJB'
```

Nótese que la función es igual a la de suma en su definición, pero se comporta diferente para cada tipo de dato, en este caso números o strings.

Con esto tenemos las herramientas básicas para intentar programar el cifrado y descifrado de algunas de las claves utilizadas en el grupo. Sin embargo, esto no pretende ser un curso de programación en Python y si no se tiene ningún conocimiento previo del tema es posible que no se pueda seguir apropiadamente el procedimiento de las páginas siguientes, por lo que se recomienda tener por lo menos algo de experiencia antes de proceder con la lectura.

1.1.1 Argumentos opcionales

En ocasiones es conveniente tener una función que tenga ciertos valores predeterminados que sean muy comúnes y no tengan que ser definidos por el usuario todo el tiempo. Supongamos que tenemos por ejemplo la siguiente función:

[10]:
```
def sumar(a,b,c):
    return a+b+c
```

Para la suma general de tres números. Si alguien solo quiere sumar 2 números, tendría que poner 0 en uno de los valores. Esto se puede arreglar preasignando 0 a la c:

[11]:
```
def sumar(a,b,c=0):
    return a+b+c
```

Así, si solo se dan los valores de a y b, para una suma de solo dos números, no tendrá que llenarse la c con un cero. Y si alguien necesita sumar los tres números, se puede poner el valor deseado y la función sirve de ambas maneras:

[12]:
```
sumar(1,2) # suma de 1 y 2 únicamente
```

[12]:
```
3
```

[13]:
```
sumar(1,2,0) # suma de 1 y 2 únicamente, poniendo c como 0 manualmente
```

[13]: 3

[14]:
```
sumar(1,2,3) # suma de 1, 2 y 3
```

[14]: 6

1.2 Input

Por medio de **input()** podemos hacer la función más interactiva con el usuario, pues permite que se escriba el valor para una variable, como un string. Por ejemplo, si quisiéramos guardar el nombre de una persona en la variable **name** lo haríamos como:

[15]:
```
name = input('Inserte su nombre: ')
```

```
Inserte su nombre:  cristeros sjb
```

El argumento de la función **input** es el texto que se despliega como descripción de lo que se va a insertar por el usuario. En este caso ponemos como instrucción que se inserte el nombre. Podemos después llamar la variable **name** y vemos que efectivamente tiene como valor el nombre que acabamos de insertar:

[16]:
```
print(name)
```

```
cristeros sjb
```

En el caso de funciones también se puede hacer. Por ejemplo para la función sumar anterior:

[17]:
```
def sumar():
    a = input('a = ')
    b = input('b = ')
    c = input('c = ')
    return a+b+c
```

La llamamos para los números 1, 2 y 4

[18]:
```
sumar()
```

```
a =  1
b =  2
c =  4
```

[18]: '124'

Pero debemos recordar que input toma valores como strings, y al sumar strings estamos concatenando solamente, no sumando los números. Para hacer la suma necesitaríamos:

[19]:
```
def sumar():
    a = int(input('a = '))
    b = int(input('b = '))
    c = int(input('c = '))
    return a+b+c
sumar()
```

```
a =  1
b =  2
c =  4
```

[19]: 7

Y obtenemos el resultado que esperaríamos. La función **int()** convierte el argumento en un número entero, por lo que convierte el string '1' a un número 1 que sí se pueda sumar como número, y no concatenar como string.

1.3 Ciclos

1.3.1 For básico

El ciclo **for** repite una acción un determinado número de veces. Por ejemplo, si queremos desplegar la palabra "cristeros" 10 veces, podríamos llamar la función print 10 veces:

[20]:
```
print('cristeros')
print('cristeros')
print('cristeros')
print('cristeros')
print('cristeros')
print('cristeros')
print('cristeros')
print('cristeros')
print('cristeros')
print('cristeros')
```

```
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
```

O podríamos utilizar un ciclo **for**:

[21]:
```
for i in range(10):
    print('cristeros')
```

```
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
cristeros
```

La función **range(num)** nos da un rango de valores del 0 al número que pongamos como entrada (num), por default. Pero se puede especificar un rango como **range(2,6)**. Este es un ejemplo más de las funciones con argumentos opcionales/predeterminados.

Podemos también desplegar cada letra de la palabra una por una:

[22]:
```
word = 'cristeros'
num = len(word)
for i in range(num):
    print(word[i])
```

```
c
r
i
s
t
e
r
o
s
```

La función **len()** nos da la longitud (cantidad de carácteres) de la palabra, para saber cuántas veces tendremos que repetir el ciclo hasta imprimir todas las letras.

1.4 For en una línea

Podemos hacer un ciclo for más compacto en casos específicos. Supongamos que queremos hacer una lista que vaya del 0 al 9. La función range nos ayuda para los for, pero no produce una lista. Sin embargo, con un for en una línea podríamos generar una lista a partir de esto:

[23]:
```
lista = [num for num in range(0,9)]
lista
```

[23]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

La palabra **num** en realidad no tiene importancia, pudo haber sido cualquier otra cosa, como roberto:

[24]:
```
lista = [roberto for roberto in range(0,9)]
lista
```

[24]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Es solo un índice y no tiene importancia cómo le pongamos. Nótese que la lista no llega hasta el 9, sino hasta el 8. Esto es porque la enumeración de Python comienza en 8 y **range(9)** nos daría un total de 9 elementos, que contando desde el cero terminan en el número 8.

1.4.1 Métodos

Los métodos son un tipo de funciones pero la manera de llamarlas es un tanto distinta. Cada tipo de objetos puede tener uno o varios métodos. Por ejemplo el método **join** de un string que sirve para unir los elementos de una lista en un string. Si tenemos por ejemplo la siguiente lista:

[4]:
```
lista = ['dar','la','vida','por','Cristo']
lista
```

[4]:
```
['dar', 'la', 'vida', 'por', 'Cristo']
```

Y queremos unirlo en una sola frase "dar la vida por Cristo". Podemos utilizar el string de espacio vacío ' ' y aplicar el método **join** para hacer esto, de la siguiente manera:

[5]:
```
frase = ' '.join(lista)
frase
```

[5]:
```
'dar la vida por Cristo'
```

Y hemos logrado lo esperado. Para llamar un método se utiliza un punto después del objeto y se llama como cualquier función normal tras esto. También está el método split para strings, que sería la operación inversa a lo que acabamos de hacer. En este caso el método es del string de la frase y no del separador, y dentro del argumento de la función colocamos el separador deseado.

[8]:
```
lista2 = frase.split(' ')
lista2
```

[8]:
```
['dar', 'la', 'vida', 'por', 'Cristo']
```

1.4.2 Método "lower"

Con el método **.lower()** podemos convertir un string a letras minúsculas. Si ponemos por ejemplo la palabra "Cristeros" y le aplicamos el método, obtendríamos "cristeros":

[48]:
```
word = 'Cristeros'
word.lower()
```

[48]: 'cristeros'

De la misma manera para todas las letras mayúsculas "CRISTEROS":

[49]:
```
word = 'CRISTEROS'
word.lower()
```

[49]: 'cristeros'

Y el método es bastante robusto, pues los demás símbolos los deja invariantes:

[55]:
```
word = '¡Cristeros 1,2,3!'
word.lower()
```

[55]: '¡cristeros 1,2,3!'

Y también pone en minúsculas carácteres especiales como la "ñ" u otras letras no comunes:

[57]:
```
word = 'ÓÍÁÚÑÜÖÄ'
word.lower()
```

[57]: 'óíáúñüöä'

Este método nos será muy útil al momento de definir nuestras funciones, pues por la naturaleza de los métodos que se utilizarán, es conveniente trabajar ya sea con puras letras mayúsculas o con puras letras minúsculas. En este caso se opta por las letras minúsculas porque es más natural escribir con ellas y se consideran más simples y amigables a la vista.

2 Contenido

Para cada una de las siguientes claves se explicará su funcionamiento y se explicará el procedimiento a seguir para idear un código que sirva para automatizar el proceso tanto de cifrado como de descifrado, siempre que sea posible. Se irán abordando en orden de dificultad/complejidad, de manera que los fundamentos utilizadas en las primeras servirán para realizar o simplificar el código para las demás. El orden es el siguiente: - Cifrado César - Cifrado Rotacional - Cifrado por Transposición - Cifrado de Vigenère - Clave Inversa - Clave Murciélago - Clave Morse

Se utilizará la convención de letras minísculas por preferencia personal, pero por lo general este tipo de claves no son "case-sensitive", i.e. no hay distinción entre letras mayúsculas y minúsculas.

Dentro de las claves que existen en el grupo y que no se abordan en este documento están la Clave Gato y la Clave Semáforo, mismas que, al ser más simbólicas/visuales, no tiene mucho sentido tratar de representarlas con código.

3 Cifrado César

Es una versión simplificada del Cifrado de Vigenère y se analizará aquí principalmente como antecedente de éste, así como para darlo a conocer mejor. Este cifrado fue uno de los primeros

métodos de cifrado y consiste en recorrer las letras del alfabeto tres letras. Es decir, las A se convierten en D (saltamos tres letras: B,C,D), las B en E, y así sucesivamente. Al llegar a la X se nos acaban las letras y comenzamos de nuevo, por lo que las X se convierten en A, las Y en B, las Z en C y comenzamos de nuevo con la A. Veamos un ejemplo. Consideremos el siguiente texto de entrada:

Entrada: cristeros

La C se convierte en F, la R se convierte en U, la I se convierte L, y así sucesivamente. Terminamos con la salida:

Salida: fulvwhurv

Esta operación parece arbitraria en primer instante. Un primer intento podría ser definir cada cambio de letra, por ejemplo que A va a D y B va a E, pero deberíamos hacer esto 25 veces (no contamos la ñ) y no sería eficiente. En vez de eso, podemos pensar en las letras como números. La A sería el 0, la B el 1, la C el 2, la D el 3, y así sucesivamente. De esta forma si le sumamos 3 a la A, que es 0, obtenemos 3, que es la letra D. Pero, ¿cómo haríamos esto?

Afortunadamente para nosotros, Python tiene una función integrada **ord()**, que convierte la letra en su valor numérico del sistema unicode. En palabras simples, cada letra está asociada a un número dentro de la computadora, y lo que es aún mejor las principales 25 letras del alfabeto (sin la ñ) están en orden, comenzando por la "a". Podemos ver esto llamando la función:

[25]:
```
ord('a')
```

[25]:
```
97
```

El número asociado a la letra "a" es el 97. Veamos el número asociado a la letra "b":

[26]:
```
ord('b')
```

[26]:
```
98
```

Como se anticipó, están en orden. Cabe mencionar que las letras mayúsculas y minúsculas son caracteres distintos, y por lo tanto no tienen el mismo número asociado la letra "a" y la "A", por ejemplo:

[27]:
```
ord('A')
```

[27]:
```
65
```

Por esta razón es que se decidió trabajar únicamente con letras minúsculas en este documento. Entonces, si restamos 97 a **ord('a')**, obtenemos un valor de a igual a 0, y si lo repetimos para b tendríamos 1, y para c 2, y así sucesivamente. Podemos definir entonces una función que pase de una letra minúscula del alfabeto a su número del 0 al 25 de la siguiente manera:

[2]:
```
def letnum(letter):
    return ord(letter) - ord('a')
```

Ahora si utilizamos la función para la letra "a" obtendríamos 0:

[29]:
```
letnum('a')
```

[29]:
```
0
```

Ahora, ¿cómo aplicamos esto para cada letra de una palabra, como la de nuestro ejemplo "cristeros"? Fácilmente, utilizando índices podemos llamar cada letra una por una. Por ejemplo, si queremos la primera:

[30]:
```
word = 'cristeros'
word[0]
```

[30]:
```
'c'
```

Hacemos un ciclo for que vaya para cada letra de la palabra:

[31]:
```
wd = [letnum(letter) for letter in word]
wd
```

[31]:
```
[2, 17, 8, 18, 19, 4, 17, 14, 18]
```

Tenemos que la "c" es el número 2, la "r" el 17, la "i" el 8 y así para el resto de las letras. Ahora le sumamos 3 a cada elemento para obtener el corrimiento del cifrado César:

[32]:
```
wd = [letnum(letter) + 3 for letter in word]
wd
```

[32]:
```
[5, 20, 11, 21, 22, 7, 20, 17, 21]
```

Tenemos ya los números de las letras, como podemos ver, pues el 5 corresponde a la f, el 20 a la u y así sucesivamente, pero esto no es fácil de leer. Necesitamos convertir esto a una palabra. Afortunadamente existe una función reversa a **ord()**, que pasa del número asociado a un caracter a ese mismo caracter. Esta función se llama **chr()**. Por ejemplo, **chr(97)** nos daría el carácter 'a'. Si queremos convertir un 0 a una a, que es lo que tenemos en este caso, debemos sumarle 97 (ord('a')) a los números para llegar a la letra correcta:

[33]:
```
wd2 = [chr(ord('a')+num) for num in wd]
wd2
```

[33]:
```
['f', 'u', 'l', 'v', 'w', 'h', 'u', 'r', 'v']
```

Tenemos ya las letras, pero están en una lista, y se sigue complicando la lectura. Utilizamos la función **''.join()** para agrupar las letras en una sola palabra:

[34]:
```
''.join(wd2)
```

[34]:
```
'fulvwhurv'
```

Y tenemos finalmente la palabra encriptada.

Ahora póngamonos en la situación opuesta. Se nos presenta la palabra "fulvwhurv" y sabemos que está en cifrado César. ¿Cómo recuperaríamos la palabra original?. Analicemos la siutación. Para cifrar con el Cifrado César, sumamos 3 a cada número de la letra. Si queremos invertir este proceso, el paso lógico sería restar 3 a cada número de la letra, en vez de restar. Por lo que repetiríamos el proceso pero en vez de sumar 3, restaríamos 3:

[35]:
```
word = 'fulvwhurv'
wd = [letnum(letter) - 3 for letter in word] # Restamos en vez de sumar
wd2 = [chr(ord('a')+num) for num in wd]
''.join(wd2)
```

[35]:
```
'cristeros'
```

Y recuperamos la palabra original. Probemos ahora con frases de más de una palabra:

Entrada: viva cristo rey

Intentemos utilizar el mismo procedimiento:

[36]:
```
word = 'viva cristo rey'
wd = [letnum(letter) + 3 for letter in word]
wd2 = [chr(ord('a')+num) for num in wd]
''.join(wd2)
```

[36]:
```
'ylyd#fulvwr#uh|'
```

Algo raro ocurre. Veamos paso a paso qué está sucediendo.

Primero capturamos la frase, no hay error ahí. Luego convertimos la frase a números:

[37]:
```
wd = [letnum(letter) + 3 for letter in word]
wd
```

[37]:
```
[24, 11, 24, 3, -62, 5, 20, 11, 21, 22, 17, -62, 20, 7, 27]
```

Nótese que tenemos dos números -62. ¿A qué se debe esto?

Recordemos que cada carácter en el string es un elemento, esto incluye los espacios, que como vemos se asocian a un número -62 (en realidad se asocian al número 32, pero tras restarle 97, que es el valor de ord('a'), y sumarle 3 llegamos a -62). Los espacios entonces estorban, por lo que sería más fácil deshacernos de ellos. Afortunadamente existe el método **split** para un string, que lo divide en palabras cuando hay más de una, separada por un espacio. En general, podemos utilizar **split(sep)** para separar el string cada vez que aparece un carácter "sep". Por default, este carácter está definido como ' ', que es el carácter de espacio. Probemos para nuestrs palabra:

[38]:
```
word.split()
```

[38]:
```
['viva', 'cristo', 'rey']
```

Ahora tenemos la frase dividida en 3 palabras y los espacios han sido eliminados, temporalmente. Nótese que **split()** es un método, no una función. La manera de llamarlo es con un punto después

del string al que lo queremos aplicar. Para investigar más sobre el tema se puede ver esta liga:

https://www.tutorialspoint.com/difference-between-method-and-function-in-python#:~:text=A%20method%20in%20python%20is,is%20contained%20within%20the%20class

Ahora que tenemos nuestras palabras divididas podemos aplicarles a cada una el método anterior con un ciclo for:

[39]:
```
word = 'viva cristo rey'
word = word.split()
wd2 = []
for i in range(len(word)):
    wd = [letnum(letter) + 3 for letter in word[i]]
    wd2 += [chr(ord('a')+num) for num in wd]+[' ']
''.join(wd2)
```

[39]:
```
'ylyd fulvwr uh| '
```

Sin embargo tenemos otro problema. La últma letra no parece ser una letra, sino un símbolo. El detalle es que, si recordamos de los números de arriba, la última letra tiene el número 27, que está fuera del rango de las letras. En la frase original tenemos una "y", y recordemos que a partir de la "x" las letras se ciclan. El número 27 en realidad debería ser un 1, para la "b". Pero no podemos restarle simplemente 26 a todos los números, pues se arruinarían los demás. Lo que podemos hacer es utilizar la operación módulo (%) que nos regresa el resudio de una división. Así, si utilizamos el módulo de 27 con respecto a 26, obtendríamos un 1, preservando el resto de los números, pues por ejemplo el módulo de 24 con respecto a 26 es nuevamente 24. Con esta operación podemos hacer que se cicle todo:

[40]:
```
word = 'viva cristo rey'
word = word.split()
wd2 = []
for i in range(len(word)):
    wd = [letnum(letter) + 3 for letter in word[i]]
    wd2 += [chr(ord('a')+num%26) for num in wd]+[' ']
''.join(wd2)
```

[40]:
```
'ylyd fulvwr uhb '
```

La operación se utiliza únicamente para el valor numérico de las letras, que deben estar siempre entre 0 y 25. Cubrimos ya todos los casos y estamos listos para crear una función general para cifrar (y descifrar) en Cifrado César.

Dado que la única diferencia entre el cifrado y descifrado es que en uno se suma y en otro se resta, podemos utilizar una función principal y llamarla dentro de otra cambiando la suma a una resta para obtener la segunda función y así ahorrarnos líneas de código. Definimos entonces las funciones add y sub para dos entradas que sume y reste dichas entradas.

[3]:
```
def add(a,b):
    return a+b
def sub(a,b):
```

```
    return a-b
```

Una vez tenidas las funciones podemos programar la función principal. Primero definimos una función que convierta las strings a un formato bold al momento de desplegarlas, por fines estéticos.

[4]:
```
def bold(string):
    return "\033[1m" + str(string) + "\033[0m"
```

Ahora creamos la función cesar que será la principal, recordando el procedimiento hecho anteriormente. La función tendrá como argumentos predeterminados un string vacío y la operación de adición.

Por último, para agregar más flexibilidad a nuestra función, y que sea más robusta, agregamos el método **lower()**, descrito anteriormente, a la palabra ingresada por el usuario. Esto permitiría al usuario ingresar letras mayúsculas sin romper el funcionamiento de nuestro código.

[5]:
```
def cesar(word='',op=add):
    wd = word.lower().split()
    wd2 = []
    for i in range(len(wd)):
        wd1 = [op(letnum(letter),3) for letter in wd[i]]
        wd2 += [chr(ord('a')+num%26) for num in wd1]
        wd2 += [' '] # Espacio entre palabras
    wd2 = ''.join(wd2)
    return wd2
```

Podemos llamar la palabra desde la función:

[128]:
```
cesar('cristeros')
```

[128]:
```
'fulvwhurv '
```

O también con la primera letra mayúscula:

[129]:
```
cesar('Cristeros')
```

[129]:
```
'fulvwhurv '
```

O con todas las letras mayúsculas:

[130]:
```
cesar('CRISTEROS')
```

[130]:
```
'fulvwhurv '
```

A partir de esto definimos nuestras dos funciones, una para cifrar y otra para descifrar.

Estas funciones deberán ser más interactivas, permitiendo la entrada de múltiples palabras consecutivas por medio de **input()**. Para esto creamos un ciclo **while**, que se ejecute mientras la palabra sea diferente a un string vacío. De esta manera, se podrán cifrar o descifrar varias frases

consecutivamente sin tener que volver a llamar la función y se podrá cerrar el ciclo simpelmente pulsando la tecla “Enter” cuando pida la palabra de entrada.

Utilizamos la función de bold que creamos para darle mejor legibilidad a la salida y utilizamos separadores de puntos para distinguir entre las diferentes entradas y salidas, como se ve a continuación:

[6]:
```
def cesar1(word=''):
    if word == '':
        word = input('Entrada: ')
    while word != '':
        wd2 = cesar(word,add)
        print("....................................................")
        print(bold("In: "),word)
        print(bold("Out:"),wd2)
        print("....................................................")
        word = input('Entrada: ')
def cesar2(word=''):
    if word == '':
        word = input('Entrada: ')
    while word != '':
        wd2 = cesar(word,sub)
        print("....................................................")
        print(bold("In: "),word)
        print(bold("Out:"),wd2)
        print("....................................................")
        word = input('Entrada: ')
```

En este caso podemos llamar la función vacía y luego escribir la palabra desde la interfaz interactiva de input. Podemos llamar entonces las funciones de la siguiente manera:

[47]:
```
cesar1()
```

```
Entrada:  cristeros

...
In:  cristeros
Out: fulvwhurv
...

Entrada:  san juan bautista

...
In:  san juan bautista
Out: vdq mxdq edxwlvwd
...

Entrada:  espiritu santo

...
In:  espiritu santo
```

```
Out: hvslulwx vdqwr
...

Entrada:
```

[48]:
```
cesar2()
```

```
Entrada:  fulvwhurv

...
In:  fulvwhurv
Out: cristeros
...

Entrada:  fpojgpoea

...
In:  fpojgpoea
Out: cmlgdmlbx
...

Entrada:
```

Tenemos finalmente las funciones útiles para el Cifrado César, todo automático.

La dinámica para el resto de las claves será la misma. Se define primero la función principal que regresa el cifrado o descifrado de una palabra o frase, y después se crean funciones específicas para cifrar y descifrar que sean más interactivas y estén pensadas para trabajar con más de una frase, siendo así más flexibles para el usuario y más útiles para un uso real, como poner ejercicios rápidamente de varias claves para una actividad, o por el contrario descifrar rápidamente los ejercicios que sean puestos.

4 Cifrado Rotacional

En el Cifrado César se recorrían las letras 3 veces, pero, ¿qué pasaría si se quiere cifrar recorriendo las letras solo 1 vez? ¿o 5 veces?

Tendríamos entonces un caso de Cifrado Rotacional. El Cifrado César es un caso especial del cifrado rotacional, con una rotación de 3 letras. En general, podríamos tener una rotación de n letras. No tenemos que armar el código de nuevo desde un principio, podemos trabajar con lo que ya hemos construido hasta ahora. En realidad, lo único que tendríamos que cambiar es el número de rotaciones, que anteriormente era 3, por un número n que el usuario pueda insertar en la función. Aprovechamos nuevamente los argumentos opcionales y dejamos por default el número 3, en honor al Cifrado César:

[7]:
```
def crot(word='',op=add,n=3):
    wd = word.lower().split()
    wd2 = []
    for i in range(len(wd)):
        wd1 = [op(letnum(letter),n) for letter in wd[i]] # Rotacíon de n letras
        wd2 += [chr(ord('a')+num%26) for num in wd1]
```

```
        wd2 += [' '] # Espacio entre palabras
    wd2 = ''.join(wd2)
    return wd2
def crot1(word='',n=3):
    if word == '':
        word = input('Entrada: ')
    while word != '':
        wd2 = crot(word,add,n=n)
        print("..........................................................")
        print(bold("In: "),word)
        print(bold("Out:"),wd2)
        print("..........................................................")
        word = input('Entrada: ')
def crot2(word='',n=3):
    if word == '':
        word = input('Entrada: ')
    while word != '':
        wd2 = crot(word,sub,n=n)
        print("..........................................................")
        print(bold("In: "),word)
        print(bold("Out:"),wd2)
        print("..........................................................")
        word = input('Entrada: ')
```

[108]:
```
crot1()
```

```
Entrada:  cristeros

...
In:  cristeros
Out: fulvwhurv
...

Entrada:  benediciencia

...
In:  benediciencia
Out: ehqhglflhqfld
...

Entrada:  esperanza

...
In:  esperanza
Out: hvshudqcd
...

Entrada:
```

Podemos ver que si llamamos la función predeterminada, con $n = 3$, obtenemos el mismo resultado que con el Cifrado César. Sin embargo, si cambiamos el valor de n:

[109]:
```
crot1(n=6)
```

```
Entrada:  cristeros

...
In:  cristeros
Out: ixoyzkxuy
...

Entrada:  benediciencia

...
In:  benediciencia
Out: hktkjoioktiog
...

Entrada:  esperanza

...
In:  esperanza
Out: kyvkxgtfg
...

Entrada:
```

Tenemos ahora un resultado diferente. Lo único que tuvimos que cambiar fue el valor de n, y nuestras funciones ya están listas.

4.1 Si no se conoce n

Para este tipo de cifrados, más generales, sería prácticamente imposible descifrar a mano un mensaje si no se sabe el valor de n. Se tendría que probar con cada uno de los valores posibles, desde el 1 hasta el 25. Afortunadamente para nosotros, podemos utilizar programación para hacer estos intentos automatizados. Podemos definir una función que nos ayude a descifrar una frase cifrada de la que no conocemos el número de rotaciones.

Supongamos que tenemos el siguiente mensaje: "aoao awo". No conocemos el número de rotaciones pero sabemos que es un mensaje cifrado con cifrado rotacional. Podemos crear un ciclo que descifre la frase para todas las n y escoger la que más tenga sentido:

[110]:
```
def crotm(word='',op=add,n=3):
    if word == '':
        word = input('Entrada: ')
    wd = word.split()
    wd2 = []
    for i in range(len(wd)):
        wd1 = [op(letnum(letter),n) for letter in wd[i]] # Rotación de n letras
        wd2 += [chr(ord('a')+num%26) for num in wd1]
        wd2 += [' '] # Espacio entre palabras
    wd2 = ''.join(wd2)
    return wd2
```

```
word = 'aoao awo'
print('"end" para salir: ')
for i in range(26):
    sal = input()
    if sal == 'end':
        break
    wd2 = crotm(word,n=i)
    print("n =",i,": ",wd2)
```

```
"end" para salir:


n = 0 :  aoao awo


n = 1 :  bpbp bxp


n = 2 :  cqcq cyq


n = 3 :  drdr dzr


n = 4 :  eses eas



n = 5 :  ftft fbt
n = 6 :  gugu gcu


n = 7 :  hvhv hdv


n = 8 :  iwiw iew


n = 9 :  jxjx jfx


n = 10 :  kyky kgy


n = 11 :  lzlz lhz
```

```
n = 12 :  mama mia

 end
```

Nos paramos en $n = 12$, porque el mensaje tiene sentido. En un futuro se podría desarrollar un algoritmo más inteligente que detecte sílabas que no existen en español y descarte la palabra, pero por el momento va más allá del alcance de este documento.

5 Cifrado por transposición

Este cifrado consiste en invertir cada palabra de la frase, i.e. la primera letra de la palabra sería la última, la segunda la penúltima, etc. Por ejemplo, si quisiéramos cifrar por transposición la frase: "dar la vida por cristo", tendríamos "rad al adiv rop otsirc". Para programar esto simplemente debemos buscar un método simple para invertir un string. Afortunadamente sí que existe. Hay una función **reversed** que funciona como un iterador inverso. No nos adentraremos tanto a lo que hace esta función en sí, sino a lo que podemos obtener de ella. Tengamos por ejemplo el siguiente texto:

[53]:
```
word = 'viva cristo rey'
word
```

[53]:
```
'viva cristo rey'
```

Lo invertimos con la función reversed:

[54]:
```
invw = reversed(word)
invw
```

[54]:
```
<reversed at 0x1bd32addbb0>
```

Y obtenemos el iterador. Este iterador se puede ver elemento a elemento con la función **next()**:

[55]:
```
print(next(invw))
print(next(invw))
print(next(invw))
print(next(invw))
print(next(invw))
print(next(invw))
print(next(invw))
```

```
y
e
r

o
t
s
```

Vamos a ir sacando los elementos uno a uno hasta completar la palabra. Para juntar todo utilizamos entonces la función join y tenemos lo siguiente:

[56]:
```
invw = reversed(word)
wd = ''.join(invw)
wd
```

[56]:
```
'yer otsirc aviv'
```

Nótese que debemos volver a establecer la definición como reverso porque al ir iterando con **next()** se van perdiendo los elementos que ya fueron puestos. Por ejemplo:

[57]:
```
invw = reversed(word)
print(next(invw))
print(next(invw))
print(next(invw))
''.join(invw)
```

```
y
e
r
```

[57]:
```
' otsirc aviv'
```

Aquí perdemos por completo la palabra rey. En general, no es necesaria la funcíon **next** para nuestros fines, por lo que evitamos llamarla de ahora en adelante. Nótese también que se invierte toda la frase, y no cada palabra. Para invertir cada palabra tendríamos que utilizar split para dividir la frase en palabras y después utilizar un for en una línea para invertir cada una, todo para finalmente unirlo todo con join.

[58]:
```
wd = word.split()
wd
```

[58]:
```
['viva', 'cristo', 'rey']
```

Ya que la dividimos la invertimos:

[59]:
```
wd2 = [''.join(reversed(elm)) for elm in wd]
wd2
```

[59]:
```
['aviv', 'otsirc', 'yer']
```

Y lo unimos con espacios entre cada palabra:

[60]:
```
wd3 = ' '.join(wd2)
wd3
```

[60]:
```
'aviv otsirc yer'
```

Ahora tenemos todo lo necesario para construir la función:

[67]:
```
def trp(word=''):
    wd = word.lower().split()
    wd2 = [''.join(reversed(elm)) for elm in wd]
    wd2 = ' '.join(wd2)
    return wd2
```

Pongamos un ejemplo:

[60]:
```
trp('hola como estan todos')
```

[60]:
```
'aloh omoc natse sodot'
```

Dado a que no se utiliza en ningún momento la conversión a números, esta clave puede aceptar prácticamente cualquier carácter:

[68]:
```
trp('¡Dónde están todos!')
```

[68]:
```
'ednód¡ nátse !sodot'
```

Sin embargo, utilizar signos de puntuación o mayúsculas al inicio de una palabra puede delatar que se trata del cifrado por transposción o escritura inversa, úsese a consciencia.

Construimos entonces la función para cifrar, pero nótese que es una clave simétrica en este aspecto, pues el cifrado se hace igual al descifrado, por lo que solo necesitamos una función:

[10]:
```
def trp1(word=''):
    if word =='':
        word = input('Texto plano: ')
    while word != '':
        wd2 = trp(word)
        print("..........................................................")
        print(bold("In: "),word)
        print(bold("Out:"),wd2)
        print("..........................................................")
        word = input('Texto plano: ')
```

En la que podemos cifrar varias frases:

[69]:
```
trp1()
```

```
Texto plano:  aleluya

...
In:  aleluya
Out: ayulela
...

Texto plano:  cristeros
```

```
...
In:  cristeros
Out: soretsirc
...

Texto plano:  viva cristo rey

...
In:  viva cristo rey
Out: aviv otsirc yer
...

Texto plano:
```

6 Cifrado de Vigenère

Este cifrado fue realmente el detonador de todo este proyecto, por lo increíblemente tedioso que es cifrar/descifrar manualmente y lo propenso que se puede ser a errores de vista o de dedo. Sin más preámbulos, el Cifrado de Vigenère se puede entender como una evolución del Cifrado Rotacional, en donde se rotaba el abecedario completo por una cantidad fija. Pues bien, en el Cifrado de Vigenère se va rotando una letra a la vez, cada una por un diferente valor numérico, dado por una clave. Es más fácil entenderlo visualmente, por medio del siguiente cuadro:

Por la entrada de texto plano se puede ver la letra que tendríamos a la entrada y por la entrada de clave sería la letra de la clave que corresponda. Se escoge una clave de una cantidad determinada de letras, por lo general no muy larga (de unas 5 letras máximo) que se estará repitiendo a lo largo de la frase. Por ejemplo, si tenemos la entrada de texto plano: "dar la vida por xto", y la clave es "rey", la clave expandida sería: "rey re yrey rey rey", que sería la letra de clave que correspondería a cada letra de la entrada de texto plano. Como se puede ver, cada letra de la clave representa en sí una rotacíon del alfabeto, al igual que en el Cifrado Rotacional. La A representa una rotación nula ($n = 0$), la B representa una rotación por 1 ($n = 1$), y así sucesivamente.

Para empezar a hacer una solución computacional podemos primero separar las palabras:

[69]:
```
word = 'dar la vida por xto'
key = 'rey'
keyN = [ord(letter)-97 for letter in key]
words = word.split()
words
```

[69]:
```
['dar', 'la', 'vida', 'por', 'xto']
```

Una vez tenido esto convertimos las palabras a letras:

[70]:
```
wd = [[]]*len(words)
for i in range(len(words)):
    wd[i] = [ord(letter) - 97 for letter in words[i]]
wd
```

[70]:
```
[[3, 0, 17], [11, 0], [21, 8, 3, 0], [15, 14, 17], [23, 19, 14]]
```

El siguiente paso sería crear una lista de las mismas dimensiones con la clave

[71]:
```
KEY = [[]]*len(words)
KEY
```

[71]:
```
[[], [], [], [], []]
```

Tenemos el arreglo vacío y tenemos que llenar cada lista con las letras de la clave que corresponden a cada palabra:

[72]:
```
ct = 0
for i in range(len(words)):
    for j in range(len(words[i])):
        KEY[i] = KEY[i] + [keyN[ct%3]]
        ct += 1
KEY
```

[72]:
```
[[17, 4, 24], [17, 4], [24, 17, 4, 24], [17, 4, 24], [17, 4, 24]]
```

Con el ciclo doble pasamos por cada letra. Tenemos además el contador global ct que nos ayuda a contabilizar el total de letras que van para saber qué letra de la clave corresponde a cada letra del texto plano.

Una vez tenido esto, sumamos los números en ambas listas.

[73]:
```
for i in range(len(words)):
    for j in range(len(words[i])):
        wd[i][j] = (wd[i][j] + KEY[i][j])%26
wd
```

[73]:
```
[[20, 4, 15], [2, 4], [19, 25, 7, 24], [6, 18, 15], [14, 23, 12]]
```

Tenemos ahora los números de las letras, solo basta pasar por cada una y convertirla de nuevo a letras:

[74]:
```
for i in range(len(words)):
    wd[i] = [chr(ord('a')+num) for num in wd[i]]
wd
```

[74]:
```
[['u', 'e', 'p'],
 ['c', 'e'],
 ['t', 'z', 'h', 'y'],
 ['g', 's', 'p'],
 ['o', 'x', 'm']]
```

Va tomando forma, necesitamos otro ciclo for para agrupar primero todas las letras de las palabras:

[76]:
```
for i in range(len(words)):
    wd[i] = ''.join(wd[i])
```

```
wd
```

[76]:
```
['uep', 'ce', 'tzhy', 'gsp', 'oxm']
```

Y finalmente unimos todas las palabras por medio de espacios:

[77]:
```
wd = ' '.join(wd)
wd
```

[77]:
```
'uep ce tzhy gsp oxm'
```

Tenemos finalmente la frase cifrada. Parecen muchos ciclos pero en realidad se pueden realizar simultáneamente algunos:

[78]:
```
word,key = 'dar la vida por xto','rey'
keyN = [ord(letter)-97 for letter in key]
words = word.split()
wd,KEY,wd2 = [[]]*len(words),[[]]*len(words),[[]]*len(words)
ct = 0
for i in range(len(words)):
    wd[i] = [ord(letter) - 97 for letter in words[i]]
    for j in range(len(words[i])):
        KEY[i] = KEY[i] + [keyN[ct%3]]
        wd[i][j] = (wd[i][j] + KEY[i][j])%26
        ct += 1
    wd2[i] = [chr(ord('a')+num) for num in wd[i]]
    wd2[i] = ''.join(wd2[i])
wd2 = ' '.join(wd2)
wd2
```

[78]:
```
'uep ce tzhy gsp oxm'
```

Y llegamos finalmente al código para la función. Nuevamente haremos una función principal y de ahí partiramos a crear las variantes para cifrado y descifrado

[11]:
```
def vig(key='',word='',op=add):
    keyN = [ord(letter) - ord('a') for letter in key] # Clave a números
    words = word.lower().split()
    wd,KEY,wd2 = [[]]*len(words),[[]]*len(words),[[]]*len(words) # Listas vacías
    ct = 0
    for i in range(len(words)):
        wd[i] = [ord(letter) - 97 for letter in words[i]] # Texto plano a números
        for j in range(len(words[i])):
            KEY[i] = KEY[i] + [keyN[ct%len(key)]] # Clave dimensiones texto plano
            wd[i][j] = op(wd[i][j],KEY[i][j])%26 # Suma/resta para cifrado/
 ↪descifrado
            ct += 1 # Conteo global
```

```
        wd2[i] = ''.join([chr(ord('a')+num) for num in wd[i]]) # num a let y␣
↪juntar letras en palabras
    wd2 = ' '.join(wd2) # Juntar palabras por medio de espacios
    return wd2
```

Para la función de cifrado utilizamos la operación add nuevamente y para la de descifrado la operación sub, ambas definidas ya previamente. Ponemos un ciclo while para repetir la función y cifrar/descifrar tantas palabras consecutivas se desee. Por lo general cuando se cifra se busca utilizar la misma clave para varias cosas, por lo que la función está diseñada para pedir la clave solo al principio y luego solo tomar como entrada el nuevo texto plano. En cambio, para la función de descifrado por lo general se tiene un solo texto y se están probando varias claves, por lo que se toma solo un texto plano y se pueden ir insertando después varias claves.

Otro detalle a considerar es que al módulo de la variable de conteo general se le cambia por un general **len(key)**, para aceptar claves de diferentes tamaños.

[43]:
```
def vig1(key="",word=""):
    if key == '':
        key = input('Clave: ')
    if word == '':
        word = input('Texto plano: ')
    while word != "":
        wd2 = vig(key,word,op=add)
        print("..........................................................")
        print(bold("In: "),word)
        print(bold("Out:"),wd2)
        print("..........................................................")
        word = input("Texto plano: ")
def vig2(key="",word=""):
    if word == '':
        word = input('Texto plano: ')
    if key == '':
        key = input('Clave: ')
    while key != "":
        wd2 = vig(key,word,op=sub)
        print("..........................................................")
        print(bold("In: "),word)
        print(bold("Out:"),wd2)
        print("..........................................................")
        key = input("Clave: ")
```

Podemos entonces hacer varios ejemplos con la misma clave REY:

[70]:
```
vig1()
```

```
Clave:  rey
Texto plano:  dar la vida por xto

...
```

```
In:  dar la vida por xto
Out: uep ce tzhy gsp oxm
...

Texto plano:  cocodrilos

...
In:  cocodrilos
Out: tsafhpzpmj
...

Texto plano:  jaguares

...
In:  jaguares
Out: aeelepvw
...

Texto plano:
```

[71]:
```
vig2()
```

```
Texto plano:  uep ce tzhy gsp oxm
Clave:  mano

...
In:  uep ce tzhy gsp oxm
Out: iec os tmtm gfb cxz
...

Clave:  rana

...
In:  uep ce tzhy gsp oxm
Out: dec cn tmhh gfp xxz
...

Clave:  miel

...
In:  uep ce tzhy gsp oxm
Out: iwl rs lvwm yoe cpi
...

Clave:  rey

...
In:  uep ce tzhy gsp oxm
Out: dar la vida por xto
...

Clave:
```

7 Clave Inversa

Esta clave consiste en una clave por sustitución. Se escribe el abecedario arriba (de izquierda a derecha, de la A a la Z), y abajo se escribe nuevamente pero de derecha a izquierda (de la Z a la A), como se ve en la siguiente tabla.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

Como se puede ver, hay simetría, la Z está arriba de la A y la A está arriba de la Z, y así para cada par de letras. Si escogemos entonces solo la primera mitad de la clave obtenemos lo siguiente:

A	B	C	D	E	F	G	H	I	J	K	L	M
Z	Y	X	W	V	U	T	S	R	Q	P	O	N

En el texto plano aplicaremos la transformación correspondiente según los pares de letras. Por ejemplo, las A se cambian por Z y las Z se cambian por A, y así para cada par. Finalmente podemos agregar la "ñ", siendo su propio par, y por tanto permaneciendo invariante ante la clave.

A	B	C	D	E	F	G	H	I	J	K	L	M	Ñ
Z	Y	X	W	V	U	T	S	R	Q	P	O	N	Ñ

Para programar esto podríamos definir los pares de letras y cambiarlos de acuerdo a esto con ayuda de un diccionario, pero existe una forma más eficiente.

Con nuestro trabajo ya realizado en el Cifrado de Vigenère podemos entender las letras como números y ver si podemos encontrar un patrón general. Nuevamente con la convención de que la A es 0 y la Z es 25, veamos si hay un patrón en las parejas:

$$a = 0, z = 25$$
$$b = 1, y = 24$$
$$c = 2, x = 23$$
$$d = 3, w = 22\vdots$$

El patrón es que siempre suman 25 las parejas. Por lo que, si tenemos una letra de una pareja, sabemos que el número de la otra será 25 menos el número que conocemos. Utilizando la función **ord()** podemos convertir una letra en número, restando 97 puesto que **ord('a')** arroja 97 y queremos empezar en cero. Entonces le restamos este número a 25 y obtenemos el número de la letra que sería pareja. A esto le sumamos 97 para utilizar la función **ord()** y regresar a la letra deseada. La operación se vería así:

$$\Delta = chr[25 - (ord(\delta) - 97) + 97]$$

Donde Δ y δ son letras pertenecientes a un par de la clave. Podemos simplificar esto:

$$\Delta = chr[25 - ord(\delta) + 97 + 97]$$

$$\Rightarrow \Delta = chr[219 - ord(\delta)]$$

Tenemos entonces la función. Probemos con una palabra:

[100]:
```
word = 'cristeros'
word = ''.join([chr(219-ord(letter)) for letter in word])
word
```

[100]:
```
'xirhgvilh'
```

Como se puede ver en la clave, la X correspnode a la C, la I corresponde a la R, la R corresponde a la I, y así sucesivamente. Nuestro código funciona. Ahora probemos para más de una palabra:

[101]:
```
word = 'dar la vida por cristo'
word = ''.join([chr(219-ord(letter)) for letter in word])
word
```

[101]:
```
'wzi»oz»erwz»kli»xirhgl'
```

Nótese que los espacios también son caracteres, y se cambian por el símbolo "»". Para arreglar esto podemos de antemano cambiar los espacios por este símbolo, dado que nuestra función es simétrica, para que al hacer la clave los vuelva a cambiar por espacios:

[105]:
```
word = 'dar la vida por cristo'
word = '»'.join(word.split(' '))
word = ''.join([chr(219-ord(letter)) for letter in word])
word
```

[105]:
```
'wzi oz erwz kli xirhgl'
```

Hemos arreglado el problema. Sin embargo, todavía hay un problema más. La lera "ñ" no es parte del alfabeto tradicional, y de hecho como veremos tiene asociado un número bastante alto:

[106]:
```
ord('ñ')
```

[106]:
```
241
```

Que es mayor a 241, y por tanto nos daría un argumento negativo para la función **ord()**, lo cual arruinaría nuestro código, pues esta función no tama argumentos negativos. Podríamos resolver esto quitando las ñ de antemano y tratando de volver a convertirlas, pero sin éxito. La mejor opción en este caso es aplicar un valor absoluto dentro de **chr()** para evitar el valor negativo de "ñ" y, como con los espacios, registrar el símbolo que sería par con la ñ y corregirlo después. Si hacemos la función para "ñ" obtenemos que es par con el siguiente símbolo:

[109]:
```
chr(abs(219-ord('ñ')))
```

[109]: '\x16'

Por lo que si insertamos una palabra con “ñ”, como “niños”, obtendríamos lo siguiente:

[111]:
```
word = 'niño'
word = '»'.join(word.split(' '))
word = ''.join([chr(abs(219-ord(letter))) for letter in word])
word
```

[111]: 'mr\x16l'

Podemos utilizar el método split en la frase para eliminar estos caracteres e insertar en su lugar, con el método join, la letra “ñ”, de la siguiente manera:

[112]:
```
word = 'niño'
word = '»'.join(word.split(' '))
word = ''.join([chr(abs(219-ord(letter))) for letter in word])
word = 'ñ'.join(word.split('\x16'))
word
```

[112]: 'mrñl'

Ahora la “ñ” se queda en su lugar y el resto de la palabra queda bien. Tenemos entonces nuestra función definida finalmente:

[73]:
```
def inv(word=''):
    wd = '»'.join(word.lower().split(' '))
    wd2 = ''.join([chr(abs(219-ord(letter))) for letter in wd])
    wd2 = 'ñ'.join(wd2.split('\x16'))
    return wd2
```

Nuevamente estamos ante una clave simétrica, por lo que solo necesitamos una función para cifrar y servirá para descifrar. Si aplicamos la función dos veces seguidas obtendremos la frase original:

[74]:
```
word = 'viva cristo rey'
print(bold('Frase original: '),word)
print(bold('Cifrado 1 vez: '),inv(word))
print(bold('Cifrado 2 veces: '),inv(inv(word)))
```

```
Frase original:  viva cristo rey
Cifrado 1 vez:  erez xirhgl ivb
Cifrado 2 veces:  viva cristo rey
```

Definimos entonces la función interactiva solamente llamando la función “inv” y agregando el ciclo while para ingresar más palabras:

[16]:
```
def inv1(word=''):
    if word == '':
        word = input('Texto plano: ')
```

```
    while word != '':
        wd = inv(word)
        print("..................................................")
        print(bold("In: "),word) # Show input word
        print(bold("Out:"),wd) # Show resulting cipher
        print("..................................................")
        word = input('Texto plano: ')
```

[75]:
```
inv1()
```

```
Texto plano:  abecedario

...
In:  abecedario
Out: zyvxvwzirl
...

Texto plano:  cobras reinas

...
In:  cobras reinas
Out: xlyizh ivrmzh
...

Texto plano:  lobos

...
In:  lobos
Out: olylh
...

Texto plano:  tigres

...
In:  tigres
Out: grtivh
...

Texto plano:
```

8 Clave Murciélago

La palabra murciélago tiene 10 letras distintas entre sí. Podemos enumerarlas del 0 al 9 de la siguiente manera:

M	U	R	C	I	E	L	A	G	O
0	1	2	3	4	5	6	7	8	9

Por lo que cada número del 0 al 9 corresponde a una letra de la palabra. Con esto en mente, la clave murciélago consiste en tomar la entrada y cambiar todas las letras de la palabra murciélago

que en ella se encuentren por los números que le corresponden según la tabla anterior. Además, si se quieren ingresar números en la clave murciélago, se sustituirían estos por las letras de la palabra, por lo que tenemos una clave simétrica como en el caso de la clave inversa, donde se cifra por pares.

Veamos un ejemplo. Supongamos que tenemos la siguiente entrada:

Entrada: dar la vida por cristo

Cambiamos las letras a por un 7, las letras l por un 6, las letras i por un 4, las letras o por un 9, las letras c por un 3 y las letras r por un 2. Obtenemos lo siguiente:

Salida: d72 67 v4d7 p92 324st9

La manera más directa de abordar esto con Python es con el uso de diccionarios. Un diccionario es una serie de pares de **keys** (claves) y **values** (valores), que se pueden entender como palabras y definiciones. Cada palabra (clave) tiene su definición (valor) y se puede llamar un valor determinado por medio de su clave. En este caso podemos definir un diccionario que relacione cada letra de la palabra murciélago (claves) con su respectivo número (valor) según la tabla. Esto se implementaría, rudimentariamente, de la siguiente manera:

[81]:
```
mur_dic = {'m':'0','u':'1','r':'2','c':'3','i':'4','e':'5','l':'6','a':'7','g':
 ↪'8','0':'9'}
```

Sin embargo, ya podemos nosotros saber de una manera más fácil que además nos dará más flexibilidad en el futuro. Definimos la palabra, tomamos cada letra de la palabra y para los números podemos utilizar un range:

[82]:
```
mur_wd = [letter for letter in 'murcielago']
mur_wd
```

[82]:
```
['m', 'u', 'r', 'c', 'i', 'e', 'l', 'a', 'g', 'o']
```

[83]:
```
mur_num = [str(num) for num in range(10)]
mur_num
```

[83]:
```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Y definimos el diccionario:

[84]:
```
mur_dic = dict(zip(mur_wd,mur_num))
mur_dic
```

[84]:
```
{'m': '0',
 'u': '1',
 'r': '2',
 'c': '3',
 'i': '4',
 'e': '5',
 'l': '6',
 'a': '7',
```

```
 'g': '8',
 'o': '9'}
```

Podrán parecer más pasos pero en realidad está más automatizado y es más fácil hacer cambios, porque en realidad existen un total de 10 claves murciélago. Se puede empezar con un 0 para la m así como se le puede asignar un 1, o un 2, o cualquier número hasta el 9. Por lo que en realidad tendríamos que definir 10 diccionarios diferentes si se hiciera con el método rudimentario, pero con nuestro método simplemente podemos sumar al momento de definir los números y agregar un módulo para asegurar que se queden entre 0 y 9:

[85]:
```
n = 3
mur_num = [str((num+n)%10) for num in range(10)]
mur_dic = dict(zip(mur_wd,mur_num))
mur_dic
```

[85]:
```
{'m': '3',
 'u': '4',
 'r': '5',
 'c': '6',
 'i': '7',
 'e': '8',
 'l': '9',
 'a': '0',
 'g': '1',
 'o': '2'}
```

De esta forma recorremos el diccionario por tres espacios, comenzando con el 3 en la m. En general, tendríamos $n = 0$, para la clave murciélago traidcional. Pero aquí vamos por todo y buscamos los métodos más robustos posibles.

Ya que tenemos el diccionario definido, es cuestión de analizar la frase de entrada letra por letra y cambiar cada letra que aparece en el diccionario por su respectivo número. Para esto podemos utilizar los índices de diccionario. Por ejemplo, el índice m nos daría (para la tradicional) un 0, que es el valor asociado a esta clave:

[86]:
```
n = 0
mur_num = [str((num+n)%10) for num in range(10)]
mur_dic = dict(zip(mur_wd,mur_num))
mur_dic['m']
```

[86]:
```
'0'
```

A diferencia de las listas y los strings que se indexan por números, aquí tenemos una clave única que se relaciona a un valor. No podemos alterar las letras de un string, por lo que tendríamos que crear una lista con las mismas dimensiones, almacenar las letras, tanto las que se quedan igual como las que se cambian a número, y después juntarlas de nuevo en una frase:

[87]:
```
word = 'dar la vida por cristo'
wd = [letter for letter in word]
print(wd)
```

```
['d', 'a', 'r', ' ', 'l', 'a', ' ', 'v', 'i', 'd', 'a', ' ', 'p', 'o', 'r', ' ',
'c', 'r', 'i', 's', 't', 'o']
```

Luego validamos con un ciclo for para cada letra si está en el diccionario, y de estarlo pondremos su valor asociado en el diccionario:

[88]:
```
for i in range(len(wd)):
    if wd[i] in mur_dic:
        wd[i] = mur_dic[wd[i]]
print(wd)
```

```
['d', '7', '2', ' ', '6', '7', ' ', 'v', '4', 'd', '7', ' ', 'p', '9', '2', ' ',
'3', '2', '4', 's', 't', '9']
```

Finalmente unimos:

[89]:
```
wd = ''.join(wd)
wd
```

[89]:
```
'd72 67 v4d7 p92 324st9'
```

Y tenemos el resultado esperado, obtenido anteriormente.

Una vez entendido esto, podemos proceder a hacer un código mucho más compacto, aprovechando las expresiones de línea sencilla. Podemos incluir la validación de las letras en el diccionario a la par que vamos pasando las letras de la palabra a la lista de letras, con un **if-else**. Si está en el diccionario, tendría el valor asignado y si no pasaríamos la pura letra. Además, podemos unir la lista en una sola pieza de texto en la misma línea. Tendríamos lo siguiente:

[90]:
```
word = 'dar la vida por cristo'
wd = ''.join([mur_dic[letter] if (letter in mur_dic) else letter for letter in␣
 ↪word])
wd
```

[90]:
```
'd72 67 v4d7 p92 324st9'
```

La sintáxis para el if de línea sencilla es:

valor-si-es-verdad ***if*** *condición* ***else*** *valor-si-es-falso*

Tendríamos así la traducción en prácticamente una línea. Nótese que al momento de definir la función, si damos la habilidad de modificar el valor de n, que afectaría la rotación de los valores de las letras, tendríamos que definir el diccionario dentro de la función. Por lo que la función quedaría:

[76]:
```
def mur(word='',n=0,key='murcielago'):
    mur_wd = [letter for letter in 'murcielago']
```

```
    mur_num = [str((num+n)%10) for num in range(10)]
    mur_dic = dict(zip(mur_wd+mur_num,mur_num+mur_wd))
    wd = ''.join([mur_dic[letter] if (letter in mur_dic) else letter for letter␣
↪in word.lower()])
    return wd
```

Esta es también una clave simétrica, por lo que aplicarla dos veces seguidas nos regresa la frase original:

[78]:
```
word = 'viva cristo rey'
print(bold('Frase original: '),word)
print(bold('Cifrado 1 vez: '),mur(word))
print(bold('Cifrado 2 veces: '),mur(mur(word)))
```

```
Frase original:  viva cristo rey
Cifrado 1 vez:  v4v7 324st9 25y
Cifrado 2 veces:  viva cristo rey
```

[85]:
```
def mur1(word='',n=0,key='murcielago'):
    if word == '':
        word = input('Texto plano: ')
    while word != '':
        wd = mur(word,n,key)
        print("..........................................................")
        print(bold("In: "),word) # Show input word
        print(bold("Out:"),wd) # Show resulting cipher
        print("..........................................................")
        word = input('Texto plano: ')
```

Definimos la opción para insertar la frase de antes o por input y después de seguir insertando frases a cifrar. Para salir del ciclo se usa la tecla enter para hacer que **word** sea igual a un string vacío y se salga del ciclo while.

[86]:
```
mur1()
```

```
Texto plano:  cristeros

...
In:  cristeros
Out: 324st529s
...

Texto plano:  viva cristo rey

...
In:  viva cristo rey
Out: v4v7 324st9 25y
...

Texto plano:  parroquia sjb
```

```
...
In:  parroquia sjb
Out: p7229q147 sjb
...

Texto plano:  pastoral juvenil

...
In:  pastoral juvenil
Out: p7st9276 j1v5n46
...

Texto plano:
```

Se pone por default la clave murciélago tradicional, pero podemos alterar el valor de n para conseguir por ejemplo la clave donde la letra M está asociada al 6:

[87]:
```
mur1(n=6)
```

```
Texto plano:  vela de maracuya

...
In:  vela de maracuya
Out: v123 d1 638397y3
...

Texto plano:  parroquia sjb

...
In:  parroquia sjb
Out: p3885q703 sjb
...

Texto plano:  cristeros

...
In:  cristeros
Out: 980st185s
...

Texto plano:
```

Y así cambian las frases cifradas anteriormente. Esto propone una ventaja enorme y hace que la función sea bastante robusta, capaz de manejar todo tipo de situación que se pueda presentar. También se puede cambiar la palabra murciélago por cualquier otra, si así se desea.

9 Clave Morse

9.1 Escrita

Llegamos a la Clave Morse, con todos nuestros conocimientos acumulados hasta ahora. No hay camino corto, tendremos que utilizar un diccionario que almacene el código morse para cada

letra, y de manera manual. Pero una vez tenido eso lo demás no será muy diferente a lo hecho ya previamente y el programa resultante valdrá la pena.

Para las letras podremos automatizar la generación de una lista de letras, pero la generación de los códigos será manual:

[26]:
```
morse_let = ['.-','-...','-.-.','-..','.','..-.','--.','....','..','.---','-.
 ↪-','.-..','--',
             '-.','---','.--.','--.-','.-.','...','-','..-','...-','.--','-..
 ↪-','-.--','--..']
alf = [chr(ord('a')+num) for num in list(range(0,26))]
print(alf)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

Después tenemos los números:

[27]:
```
morse_num = ['-----','.----','..---','...--','....-','.....','-....','--...
 ↪','---..','----.']
nums = [str(num) for num in range(0,10)]
nums
```

[27]:
```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Ahora podemos definir el diccionario con puras letras y números:

[28]:
```
morse_dic = dict(zip(alf+nums,morse_let+morse_num))
morse_dic
```

[28]:
```
{'a': '.-',
 'b': '-...',
 'c': '-.-.',
 'd': '-..',
 'e': '.',
 'f': '..-.',
 'g': '--.',
 'h': '....',
 'i': '..',
 'j': '.---',
 'k': '-.-',
 'l': '.-..',
 'm': '--',
 'n': '-.',
 'o': '---',
 'p': '.--.',
 'q': '--.-',
 'r': '.-.',
 's': '...',
```

```
 't': '-',
 'u': '..-',
 'v': '...-',
 'w': '.--',
 'x': '-..-',
 'y': '-.--',
 'z': '--..',
 '0': '-----',
 '1': '.----',
 '2': '..---',
 '3': '...--',
 '4': '....-',
 '5': '.....',
 '6': '-....',
 '7': '--...',
 '8': '---..',
 '9': '----.'}
```

Después definimos los símbolos más exóticos y los signos de puntuación en un arreglo en forma de tuplas, para reducir el riesgo de cometer error al transcribir cada código.

[29]:
```
morse_sym = [('á','.--.-'),('é','..-..'),('ó','---.'),('ä','.-.-'),('ö','---.
↪'),('ü','..--'),('à','.--.-'),
             ('è','.-..-'),('ç','-.-..'),('Ð','..--.'),('Ĝ','--.-.'),('Ĵ','.---.
↪'),('Š','...-.'),
             ('Þ','.--..'),('ß','...--..'),('&','.-...'),('.','.-.-.
↪-'),(',','--..--'),(';','-.-.-.'),
             (':','---...'),('¿','..-.-'),('?','..--..'),('¡','--...-'),('!','-.
↪-.--'),("'",'.----.'),
             ('"','.-..-.'),('(','-.--.'),(')','-.--.-'),('/','-..-.'),('-','-..
↪..-'),('_','..--.-'),
             ('+','.-.-.'),('=','-...-'),('$','...-..-'),('@','.--.-.
↪'),('ñ','--.--'),(' ','')]
```

Estos símbolos son los que estaban a nuestra disposición, pero si se requieren menos o más se puede modificar al gusto. En lo personal, consideraría que es mejor tener más para que la función sea más robusta y no falle con algunos carácteres. Tenemos como último carácter especial el espacio, que traducimos como un string vacío. Por ahora parece no tener sentido, pero más adelante se verá la importancia de esto.

Definimos ahora una función para agregar cada una de estas tuplas como elementos del diccionario:

[30]:
```
def addic(tup,dic=morse_dic):
    dic[tup[0]] = tup[1]
```

Y la aplicamos:

[31]:
```
for i in range(len(morse_sym)):
    addic(morse_sym[i])
```

Podemos también agregar las indicaciones especiales:

[32]:
```
morse_sen = [('R','...-.'),('E','........'),('T','-.-'),('W','.-...'),
             ('M','.-.-.'),('F','...-.-'),('B','-.-.-'),('A','-')]
# R - roger
# E - error
# T - invitación a transmitir
# W - esperar
# M - fin de mensaje
# F - fin de transmisión
# B - comienzo de transmisión
# A - atención
for i in range(len(morse_sen)):
    addic(morse_sen[i])
morse_dic
```

[32]:
```
{'a': '.-',
 'b': '-...',
 'c': '-.-.',
 'd': '-..',
 'e': '.',
 'f': '..-.',
 'g': '--.',
 'h': '....',
 'i': '..',
 'j': '.---',
 'k': '-.-',
 'l': '.-..',
 'm': '--',
 'n': '-.',
 'o': '---',
 'p': '.--.',
 'q': '--.-',
 'r': '.-.',
 's': '...',
 't': '-',
 'u': '..-',
 'v': '...-',
 'w': '.--',
 'x': '-..-',
 'y': '-.--',
 'z': '--..',
 '0': '-----',
 '1': '.----',
```

```
'2': '..---',
'3': '...--',
'4': '....-',
'5': '.....',
'6': '-....',
'7': '--...',
'8': '---..',
'9': '----.',
'á': '.--.-',
'é': '..-..',
'ó': '---.',
'ä': '.-.-',
'ö': '---.',
'ü': '..--',
'à': '.--.-',
'è': '.-..-',
'ç': '-.-..',
'Đ': '..--.',
'Ĝ': '--.-.',
'Ĵ': '.---.',
'Š': '...-.',
'Þ': '.--..',
'ß': '...--..',
'&': '.-...',
'.': '.-.-.-',
',': '--..--',
';': '-.-.-.',
':': '---...',
'¿': '..-.-',
'?': '..--..',
'¡': '--...-',
'!': '-.-.--',
"'": '.----.',
'"': '.-..-.',
'(': '-.--.',
')': '-.--.-',
'/': '-..-.',
'-': '-....-',
'_': '..--.-',
'+': '.-.-.',
'=': '-...-',
'$': '...-..-',
'@': '.--.-.',
'ñ': '--.--',
' ': '',
'R': '...-.',
'E': '...',
```

```
 'T': '-.-',
 'W': '.--...',
 'M': '.-.-.',
 'F': '...-.-',
 'B': '-.-.-',
 'A': '-'}
```

Tenemos finalmente el diccionario completo. Para traducir de texto plano a morse bastaría un simple for de una línea para unir las palabras y tenemos lo siguiente:

[33]:
```
word = 'viva cristo rey'
wd = [morse_dic[letter] for letter in word]
print(wd)
```

```
['...-', '..', '...-', '.-', '', '-.-.', '.-.', '..', '...', '-', '---', '',
'.-.', '.', '-.--']
```

Ahora unimos las letras:

[34]:
```
wd2 = ''.join(wd)
wd2
```

[34]:
```
'...-...-.--.-..-...----.-..-.--'
```

Necesitamos agregar un separador, en el caso de morse se acostumbra utilizar el slash (/):

[35]:
```
word = 'viva cristo rey'
wd = [morse_dic[letter] + '/' for letter in word]
wd2 = ''.join(wd)
wd2
```

[35]:
```
'...-/../.../.-//-.-./.-./../.../-/---//.-././-.--/'
```

Lo agregamos después de cada letra, y nótese como entre cada palabra hay doble diagonal, como se supone debería ser. Esto se debe a que los espacios blancos siguen siendo un elemento de la lista que contiene las letras al momento de traducirlas a morse, por lo que se agrega una diagonal extra en esos casos. Tenemos entonces lo suficiente para hacer una función que traduzca a morse:

[80]:
```
def morse(word=''):
    wd = [morse_dic[letter] + '/' for letter in word.lower()]
    wd2 = ''.join(wd)
    return wd2
```

Veamos un ejemplo:

[37]:
```
morse('dar la vida por xto')
```

[37]:
```
'-../.-/.-.//.-../.-//...-/../-../.-//.--./---/.-.//-..-/-/---/'
```

Ahora hagamos la función interactiva:

[83]:
```
def morse1(word=''):
    if word == '':
        word = input('Texto plano: ')
    while word != '':
        wd2 = morse(word)
        print("..........................................................")
        print(bold("In: "),word) # Show input word
        print(bold("Out:"),wd2) # Show resulting cipher
        print("..........................................................")
        word = input('Texto plano: ')
```

Podremos ingresar varias frases seguidas:

[84]:
```
morse1()
```

```
Texto plano:  dar la vida por xto

...
In:  dar la vida por xto
Out: -../.-/.-.//.-../.-//.../../-../.-//.--./---/.-.//-..-/-/---/
...

Texto plano:  amor

...
In:  amor
Out: .-/--/---/.-./
...

Texto plano:  rinocerontes

...
In:  rinocerontes
Out: .-./../-./---/-.-././.-./---/-./-/./.../
...

Texto plano:  nados

...
In:  nados
Out: -./.-/-../---/.../
...

Texto plano:
```

Otra funcionalidad útil sería poder traducir de morse de vuelta al texto plano, i.e. descifrar. Para esto utilizaremos un nuevo diccionario invertido:

[39]:
```
morse_dic_inv = dict(zip(morse_let+morse_num,alf+nums))
def addic_inv(tup,dic=morse_dic_inv):
    dic[tup[1]] = tup[0]
```

```
for i in range(len(morse_sym)):
    addic_inv(morse_sym[i])
morse_dic_inv
```

[39]:
```
{'.-': 'a',
 '-...': 'b',
 '-.-.': 'c',
 '-..': 'd',
 '.': 'e',
 '..-.': 'f',
 '--.': 'g',
 '...': 'h',
 '..': 'i',
 '.---': 'j',
 '-.-': 'k',
 '.-..': 'l',
 '--': 'm',
 '-.': 'n',
 '---': 'o',
 '.--.': 'p',
 '--.-': 'q',
 '.-.': 'r',
 '...': 's',
 '-': 't',
 '..-': 'u',
 '...-': 'v',
 '.--': 'w',
 '-..-': 'x',
 '-.--': 'y',
 '--..': 'z',
 '-----': '0',
 '.----': '1',
 '..---': '2',
 '...--': '3',
 '....-': '4',
 '.....': '5',
 '-....': '6',
 '--...': '7',
 '---..': '8',
 '----.': '9',
 '.--.-': 'à',
 '..-..': 'é',
 '---.': 'ö',
 '.-.-': 'ä',
 '..--': 'ü',
 '.-..-': 'è',
 '-.-..': 'ç',
```

```
    '..--.': 'Ð',
    '--.-.': 'Ĝ',
    '.---.': 'Ĵ',
    '...-.': 'Š',
    '.--..': 'Þ',
    '...--..': 'ß',
    '.-...': '&',
    '.-.-.-': '.',
    '--..--': ',',
    '-.-.-.': ';',
    '---...': ':',
    '..-.-': '¿',
    '..--..': '?',
    '--...-': '¡',
    '-.-.--': '!',
    '.----.': "'",
    '.-..-.': '"',
    '-.--.': '(',
    '-.--.-': ')',
    '-..-.': '/',
    '-....-': '-',
    '..--.-': '_',
    '.-.-.': '+',
    '-...-': '=',
    '...-..-': '$',
    '.--.-.': '@',
    '--.--': 'ñ',
    '': ' '}
```

Y seguimos la misma lógica:

[40]:
```
def morsi(word=''):
    wd = word.lower().split('/')
    wd = [morse_dic_inv[letter] for letter in wd]
    wd2 = ''.join(wd)
    return wd2
```

[41]:
```
morsi('-../.-/.-.//.-../.-//...-/../-../.-//.--./---/.-.//-..-/-/---/')
```

[41]:
```
'dar la vida por xto '
```

Probando con el morse obtenido de la traducción de texto plano a morse vemos que funciona bien.

Con esto hacemos la versión interactiva para el descifrado de morse. Nótese que aquí tenemos que definir dos funciones completamente diferentes para el cifrado y descifrado, pues no es cuestión de cambiar símbolos, sino de generar unos nuevos con diferentes reglas de espaciado entre letras y palabras.

[42]:
```
def morse2(word=''):
    if word == '':
        word = input('Texto plano: ')
    while word != '':
        wd2 = morsi(word)
        print("..........................................................")
        print(bold("In: "),word) # Show input word
        print(bold("Out:"),wd2) # Show resulting cipher
        print("..........................................................")
        word = input('Texto plano: ')
```

[88]:
```
morse2()
```

```
Texto plano:  .../---/...

...
In:  .../---/...
Out: sos
...

Texto plano:  .-/-..././.---/.-

...
In:  .-/-..././.---/.-
Out: abeja
...

Texto plano:  .-./../-./---/...

...
In:  .-./../-./---/...
Out: rinos
...

Texto plano:
```

9.2 Dictado

Para hacer un dictado de morse, generando archivos de audio que reproduzcan lo en mensaje que queramos transmitir, podemos utilizar la librería de **pydub**. Se debe instalar la librería y además instalar **simpleaudio**. Se pueden instalar ambos desde la libreta jupyter:

!pip install pydub

!pip install simpleaudio

Pero aquí no se hará porque ya se tienen instalados. De este paquete se importarán solamente un par de funciones. La primera de ellas es **AudioSegment**, que nos ayuda a generar audio y silencios. La segunda es **play**, que nos ayuda a reproducir audio, y la tercera es **Sine**, que nos ayuda a generar una onda senoidal de sonido, que en términos simples es un "beep" de una sola frecuencia, ideal para lo que queremos lograr aquí. Importamos entonces estas funciones:

[95]:
```
from pydub import AudioSegment
from pydub.playback import play
from pydub.generators import Sine
```

Ahora creemos una onda senoidal con la función **Sine**. Para fines de este documento se piensa que la frecuencia de 700Hz está en el rango cómodo para el oído, por lo que se usa esta frecuencia para generar la onda.

[96]:
```
gen = Sine(700)
gen
```

[96]:
```
<pydub.generators.Sine at 0x1d10aac23d0>
```

De momento tenemos solo un generador de onda. Para generar un sonido o un audio tenemos que convertirlo a un segmento de audio con el método **.to_audio_segment(duration)**, con la duración en milisegundos. Hagamos una prueba:

[97]:
```
test = gen.to_audio_segment(duration=1000)
test
```

[97]:
```
<pydub.audio_segment.AudioSegment at 0x1d10aac25e0>
```

En la libreta de jupyter se puede reproducir el audio, que es un sonido a 700Hz con una duración de un segundo. También podemos automatizar esto con la función play:

[98]:
```
play(test)
```

Que reproducirá el audio una vez. Teniendo esto pasamos a lo siguiente. La idea es generar una señal de audio para cada carácter en morse: para el punto, para la raya, para los espacios entre letras, para los espacios entre palabras, y también para los espacios entre puntos y rayas dentro de una misma letra. Recordemos las siguientes equivalencias:

- 1 raya = 3 puntos
- 1 diagonal = 3 puntos
- 1 doble diagonal = 7 puntos

Además el espacio entre puntos/rayas dentro de cada letra es igual a un punto, por lo que conviene definir una unidad base como tiempo de duración del punto y a partir de ahí construir las duraciones para los demás. Definimos entonces lo siguiente:

[99]:
```
unit = 400
dot = gen.to_audio_segment(duration=unit)
dash = gen.to_audio_segment(duration=3*unit)
p = AudioSegment.silent(duration=unit) # Espacio dentro de letra
pp = AudioSegment.silent(duration=3*unit) # Espacio entre letras
ppp = AudioSegment.silent(duration=7*unit) # Espacio entre palabras
```

Para este caso utilizamos un punto de 400 milisegundos para que el ritmo del dictado sea sencillo

de seguir. Lo siguiente sería construir un diccionario que relacione cada símbolo de morse (.,-,/,//) con su respectivo sonido. Llamamos a este diccionario MAD (Morse Audio Dictionary):

[100]:
```
MAD = {'.':dot,'-':dash,'0':p,'/':pp,'//':ppp} # Morse Audio Dictionary (MAD)
```

Ahora solo necesitamos probarlo para un mensaje de morse. Tomamos uno de los mensajes anteriores:

[101]:
```
morse_word = '.../../.../.-//-.-./.-./../.../-/---//.-././-.--/'
morse_audio = [MAD[sym] for sym in morse_word]
morse_audio
```

[101]:
```
[<pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
 <pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
 <pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
```

```
<pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
<pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
<pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
<pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
<pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
<pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
<pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
<pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
<pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
<pydub.audio_segment.AudioSegment at 0x1d10aab7d60>,
<pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
<pydub.audio_segment.AudioSegment at 0x1d10aac2d00>,
<pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
<pydub.audio_segment.AudioSegment at 0x1d10aacd3a0>,
<pydub.audio_segment.AudioSegment at 0x1d10aab7d60>]
```

Tenemos entonces todos los segmentos de audio, pero cada uno individualmente. Si queremos juntar todo esto en una sola pieza de audio podemos simplemente sumar. Dentro de este paquete **pydub** se puede utilizar la operación de suma para juntar piezas de audio en una sola. Así que utilizamos la función **sum(list)** que nos dará la suma de toda la lista de piezas individuales:

[102]:
```
morse_fin = sum(morse_audio)
morse_fin
```

[102]:
```
<pydub.audio_segment.AudioSegment at 0x1d10aac2670>
```

Y tenemos nuestro archivo de audio. Este es un archivo temporal y no está guardado en ningún lado, si queremos guardarlo necesitaríamos una función adicional. En realidad es un método, llamado **export**, que nos permite exportar el archivo de audio a un archivo dentro de la carpeta donde esté nuestra libreta de Jupyter con un formato determinado. Para fines de este documento utilizamos el formato mp3, por ser el más compacto. Exportamos nuestro audio como "test1.mp3" y veremos reflejado un nuevo archivo con ese nombre dentro de la carpeta:

[103]:
```
morse_fin.export('test1.mp3',format = 'mp3')
```

[103]:
```
<_io.BufferedRandom name='test1.mp3'>
```

Si se desea exportar un archivo a una carpeta en específico se puede utilizar el "path" completo de la carpeta. El path de una carpeta se vería como algo así: "C:", o en cualqueir otro idioma, según sea el caso.

Tenemos entonces todo lo necesario para hacer una función que reciba un texto plano y nos genere un archivo de audio con su dictado de morse correspondiente. Pero antes, necesitamos definir una función de morse menos interactiva que solo nos regrese la clave morse y no la imprima, para tener el texto en morse que se utilizará para convertir en audio:

Ahora definimos la función final:

[104]:
```
def morsd(word='',freq=700,unit=350,file=False):
    if word == '':
        word = input('Texto plano: ')
    gen = Sine(freq)
    dot = gen.to_audio_segment(duration=unit)
    dash = gen.to_audio_segment(duration=3*unit)
    p = AudioSegment.silent(duration=unit)
    pp = AudioSegment.silent(duration=3*unit)
    ppp = AudioSegment.silent(duration=7*unit)
    MAD = {'.':dot,'-':dash,'0':p,'/':pp,'//':ppp}
    wd = morse(word)
    morseplit = '0'.join([sym for sym in wd])
    morseplay = [MAD[sym] for sym in morseplit]
    out = sum(morseplay)
    if file:
        file_name = input('Nombre del archivo: ')
        out.export(file_name+'.mp3',format = 'mp3')
    return out
```

Podemos hacer un ejemplo con la palabra abecedario, generando un archivo y llamándolo de la misma manera, cambiando la variable file a True.

[105]:
```
morsd(file=True)
```

```
Texto plano:  abecedario
Nombre del archivo:  abecedario
```

[105]:
```
<pydub.audio_segment.AudioSegment at 0x1d10aab7d00>
```

Hagamos un ejemplo normal con la palabra ‘cristeros’:

[149]:
```
morsd('cristeros')
```

[149]:
```
<pydub.audio_segment.AudioSegment at 0x1bd32a3c670>
```

En el archivo PDF podrá no verse, pero se genera una pequeña interfaz que permite ver la duración del audio, controlar el volumen, reproducirlo y navegar por su duración. De esta forma, se puede interactuar con el audio desde la libreta de Jupyter, o bien se puede exportar el audio para utilizarlo en otra interfaz o para otros fines.

Podemos adicionalmente definir una función con un dictado formal, que comience con la llamada de atención y termine con el fin de mensaje, para una transmisión unidireccional simple:

[106]:
```
def morse_tr(word='',freq=700,unit=350):
    if word == '':
        word = input('Texto plano: ')
    word = 'A ' + word + ' M'
    gen = Sine(freq)
    dot = gen.to_audio_segment(duration=unit)
```

```
    dash = gen.to_audio_segment(duration=3*unit)
    p = AudioSegment.silent(duration=unit)
    pp = AudioSegment.silent(duration=3*unit)
    ppp = AudioSegment.silent(duration=7*unit)
    MAD = {'.':dot,'-':dash,'0':p,'/':pp,'//':ppp}
    wd = morse(word)
    morseplit = '0'.join([sym for sym in wd])
    morseplay = [MAD[sym] for sym in morseplit]
    return sum(morseplay)
morse_tr('cristeros')
```

[106]: `<pydub.audio_segment.AudioSegment at 0x1d10a8ad520>`

Con esto terminamos nuestras funciones de clave morse. En un futuro se podría intentar crear también una función que reciba un audio como dictado de morse y lo descifre a una palabra, pero ese es un reto para otro día.

Se puede tambien aprovechar las funciones de multithreading del paquete threading para hacer una función que sirva como dictado de práctica para morse, generando una palabra aleatoria de n cantidad de letras y reproduciendo el audio para que el usuario inserte entonces la palabra que acaba de escuchar y reciba entonces retroalimentación sobre si lo hizo bien o mal. Esta función, por su naturaleza, es un poco más complicada de explicar y por tanto se deja como ejercicio para el lector el entender cómo funciona.

[107]:
```
import string,random,threading
def practice(n = 10,unit=400):
    word = ''.join(random.choice(string.ascii_lowercase) for _ in range(n))
    wd2 = morsd(word,unit=unit)
    def f1():
        answer = input('Frase dictada:')
        if answer == word:
            print('¡Correcto!')
        else:
            print('Incorrecto')
            print('La frase era:',word)
    def f2(aud):
        play(aud)
    threading.Thread(target=f1).start()
    threading.Thread(target=f2(wd2)).start()
```

10 Conclusiones

Este pequeño gran proyecto sirve como demostración de que en Cristeros hay demasiado material como para dejar de aprender, incluso por la edad. Si los miembros del grupo realmente tienen una motivación para aprender y aspiran a crecer como personas, el grupo provee suficientes actividades, retos y conocimientos para mantenerlos motivados. Sin embargo, esta motivación a veces no viene fácil, y debemos ayudar a motivar a los miembros con actividades interesantes y

buscando apoyarlos en lo que les apasiona. Este proyecto personal se puede utilizar para muchos fines y como se vio, tiene aun varias áreas de mejora, tanto en la optimización de las funciones, como en la creación de funciones más generales que puedan hacer un poco más de lo que se pudo hacer aquí. Cabe mencionar también el poder del lenguaje Python y sus muy diversas implementaciones. En el grupo de Cristeros hay lugar para todo tipo de personas, solo es cuestión de encontrarlo.

11 Material Externo

El archivo de esta libreta de Jupyter (.pynb) así como un concentrado de las funciones en un código (.py) se pueden encontrar en el siguiente repositorio de GitHub

https://github.com/DiegoRdz99/Cristeros