1 Listas

En Python existen tipos de datos que se componen de otros datos. Entre estos, se encuentran las **listas**.

Las listas son un tipo de dato que permiten almacenar un grupo de datos de distintos tipo. Por ejemplo: números enteros, flotantes, caracteres, strings e incluso otras listas.

Las listas se declaran utilizando corchetes (`[ ]`) y separando cada uno de los elementos por una coma (`,`).

A continuación se muestra un ejemplo de una lista que almacena tres valores enteros 1, 2 y 3:

[3]:
```
lista_numerica = [1, 2, 3]
print(lista_numerica)
```

```
[1, 2, 3]
```

Del mismo modo, es posible guardar `strings` dentro de una lista:

[4]:
```
lista_palabras = ["Hola", "Holi", "Holiwi"]
print(lista_palabras)
```

```
['Hola', 'Holi', 'Holiwi']
```

En el caso de que dentro de la lista se agreguen operaciones o cálculos, Python primero **evaluará** cada operación por separado, para luego almacenar una lista con los resultados:

[5]:
```
lista_resultados = [2*4, 34//3, int(3.4)]
print(lista_resultados)
```

```
[8, 11, 3]
```

Dependiendo del problema a resolver, a veces será necesario inicializar listas sin elementos, es decir, **listas vacías**. Para ello se declara la lista utilizando solamente corchetes, como se ejemplifica aquí:

[6]:
```
lista_vacia = []
print(lista_vacia)
```

```
[]
```

En particular las listas vacías se usan para luego ir agregándole elementos dentro de un programa.

Cabe destacar que, a nivel interno para Python, una lista es solo una colección ordenada de elementos, por lo que estos no tienen que tener el mismo tipo:

[7]:
```
lista_ejemplo = [1, 1.2, "hola"]
print(lista_ejemplo)
```

```
[1, 1.2, 'hola']
```

1.1 Indexación

Al igual que los strings, es posible acceder a los elementos de una lista usando sus **índices**.

En el caso de las listas cada uno de sus elementos posee un índice partiendo desde el 0 hasta $len(lista) - 1$. Por ejemplo, la lista_posiciones, que tiene los elementos del 100 al 108 y un $len(lista\_posiciones) = 9$.

[8]:
```
lista_posiciones = [100, 101, 102, 103, 104, 105, 106, 107, 108]
```

Por lo tanto, tiene sus elementos indexados desde el 0 al 8. A modo de ejemplo, si quisiera acceder al elemento en la **posición 5** de la lista, o el valor 105, debería acceder del siguiente modo:

[9]:
```
# Se declara la lista con sus respectivos elementos
lista_posiciones = [100, 101, 102, 103, 104, 105, 106, 107, 108]

# Se accede a la posición deseada (en este caso la posición 5)
lista_posiciones[5]
```

[9]:
```
105
```

Es decir, al utilizar la sentencia:

`lista[indice]`

Puedo acceder a cada elemento de la lista.

IMPORTANTE: Uno de los errores más comunes al trabajar con listas es el error

`IndexError: list index out of range`

el cual aparece cuando estoy intentando acceder a una posición más allá de los elementos de la lista. Por ejemplo:

[10]:
```
lista_posiciones[9]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[10], line 1
----> 1 lista_posiciones[9]

IndexError: list index out of range
```

Una de las propiedades que hace que las listas sean de gran utilidad a la hora de programar es su **mutabilidad**, lo que significa que puedo modificar la lista sin tener que sobreescribirla.

Esto nos permite agregarle, quitarle o modificar elementos, sin tener que borrar la lista entera y guardarla nuevamente en una variable.

Por ejemplo, si queremos **modificar** o **actualizar** un elemento de la lista, el operador **asignación** (=) que hemos usado para declarar variables, nos permite modificar elementos de la lista:

[12]:
```
# Se declara una lista con valores
lista_ejemplo = [1, 2, 3, 4, 5]

# Se accede a la posición deseada y se le asigna un nuevo valor
# En este caso, la "variable asignada" es la lista en la posición deseada
lista_ejemplo[2] = 100

# Se muestra por pantalla la lista con su valor actualizado
print(lista_ejemplo)
```

```
[1, 2, 100, 4, 5]
```

Al igual que al consultar por un elemento de la lista, si queremos modificar algun valor, este debe existir, ya que, de no ser así, Python también entregará el error `IndexError`, solo que con el mensaje `list assignment index out of range`, o alguna variación, indicándonos que queremos modificar una posición que no existe realmente en la lista.

[13]:
```
# Se declara una lista de valores
lista_ejemplo2 = [1, 2, 3, 4, 5]

# Se accede a la posición deseada para asignar un nuevo valor
# Pero la posición deseada está más allá de lo que tiene la lista
lista_ejemplo2[7]= 8

# Se muestra por pantalla la lista con su valor actualizado
print(lista_ejemplo2)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[13], line 6
      2 lista_ejemplo2 = [1, 2, 3, 4, 5]
      4 # Se accede a la posición deseada para asignar un nuevo valor
      5 # Pero la posición deseada está más allá de lo que tiene la lista
----> 6 lista_ejemplo2[7]= 8
      8 # Se muestra por pantalla la lista con su valor actualizado
      9 print(lista_ejemplo2)

IndexError: list assignment index out of range
```

Cuando accedemos al elemento en la posición que tomamos, estamos utilizando esto como *variable* que almacena el valor, por lo que podemos hacer todo lo que normalmente hacemos con el valor directamente con la lista en la posición seleccionada:

[71]:
```
lista_ejemplo = [30, 40.0, "Hola"]
# Los primeros dos valores son  números, así que puedo utilizarlos como tal
```

```
valor = 2*lista_ejemplo[0] + lista_ejemplo[1] % 3

# Y el elemento 2 es un string, por lo que puedo acceder a los métodos de string
# directamente con él
if lista_ejemplo[2].isalpha():
    print(valor, lista_ejemplo[2].upper())
else:
    print(lista_ejemplo[2].isdigit())
```

```
61.0 HOLA
```

1.2 Operaciones sobre Listas

Python posee múltiples operaciones, métodos y funciones para utilizar sobre el objeto `list`. Si bien, no es el objetivo del curso conocerlas todas, a continuación se mencionan algunas de las más importantes: 1. Función `list()` 2. Agregar elementos 3. Obtener la cantidad de elementos 4. Cambiar/actualizar un valor 5. Eliminar elementos 6. Cortar (*Slicing*) 7. Concatenar 8. Repetir 9. Cambiar varios valores

1.2.1 1. Funcion `list()`

Una forma de crear, o definir, una lista vacía es utilizando la función nativa `list()`. La función `list()` recibe como entrada un valor que debe ser "iterable" (por ejemplo, otra lista o un string) para poder generar una lista. Es otras palabras, es una función de cambio de tipo.

[15]:
```
# Genera una lista vacía
lista_ejemplo = list()
print(lista_ejemplo)
```

```
[]
```

Debe destacarse que, aunque es válido usar la función `list()` sin parámetros para generar una lista vacía, como se ve en el ejemplo, esto se considera **mala práctica**, pues los estándares dicen que se debe preferir los corchetes vacíos, `[]`.

[16]:
```
lista_letras = list("Hola mundo")
print(lista_letras)
```

```
['H', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o']
```

Del mismo modo, se observa que, al ingresar un *string* (elemento iterable) a la función `list()`, este genera una lista donde cada elemento de ella es la mínima unidad en la que se puede dividir un `string` (caracteres).

Si intento utilizar la función de cambio de tipo sobre un elemento que no es iterable, obtendré un error, pues Python no sabe como resolver esta conversión. Por ejemplo:

[17]:
```
# Un número no es iterable
lista = list(10)
print(lista)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 2
      1 # Un número no es iterable
----> 2 lista = list(10)
      3 print(lista)

TypeError: 'int' object is not iterable
```

Para entender esto, tenemos que considerar algo importante: si bien para nosotros un número está conformado por *dígitos*, en un computador, un número es un "átomo", es decir, es indivisible, pues es una secuencia precisa de bits que representan exactamente a ese número en el sistema numérico binario. Es decir, lo que para nosotros es una secuencia de dígitos, para el computador es **otra** secuencia de dígitos. Por lo tanto, para evitar controversia, el número no se descompone y, por lo tanto, no se puede iterar.

1.2.2 2. Agregar Elementos

Para poder agregar elementos a una lista, Python provee de un **método** denominado `.append()`, el cual agrega un elemento al final de la lista que lo usa.

[21]:
```
lista_numeros = [1, 2, 3]
print("Antes de append (a.A.):", lista_numeros)
lista_numeros.append(4)
print("Después de append (d.A.):", lista_numeros)
```

```
Antes de append (a.A.): [1, 2, 3]
Después de append (d.A.): [1, 2, 3, 4]
```

[20]:
```
lista_numeros_2 = [1, 2, 3]
print("a.A.:", lista_numeros_2)
lista_numeros_2.append("hola_mundo")
print("d.A.:", lista_numeros_2)
```

```
a.A.: [1, 2, 3]
d.A.: [1, 2, 3, 'hola_mundo']
```

Vale la pena recordar que los **métodos** dependen del objeto sobre el cual se está trabajando. En particular, el método `.append()` modifica internamente la lista, por lo que no devuelve nada.

Un error común es asignar la lista nuevamente al resultado de `.append()`, sin embargo, esto genera que la lista completa sea eliminada, como se muestra a continuación:

[22]:
```
lista_numeros_2 = [1, 2, 3]
print("a.A.:", lista_numeros_2)
lista_numeros_2 = lista_numeros_2.append("hola_mundo")
print("d.A.:", lista_numeros_2)
```

```
a.A.: [1, 2, 3]
d.A.: None
```

En este caso, al usar `print()`, vemos que retorna `None`, esto nos está indicando que, en la variable `lista_numeros_2`, no existe **nada**, pues el uso incorrecto de `.append()` eliminó toda la lista.

1.2.3 3. Obtener la cantidad de elementos

Como ya vimos anteriormente, para los tipos de datos compuestos, existe la función nativa `len()`, que nos retorna la cantidad de elementos que posee cada objeto.

Cuando lo usamos con las listas, nos indica la cantidad de elementos que esta posee:

[23]:
```
lista_vacia = []
print(len(lista_vacia))
```

```
0
```

Una lista vacía, al no tener elementos, su "largo" es 0.

[24]:
```
lista_numeros = [1, 2, 3]
print(len(lista_numeros))
```

```
3
```

Si los elementos de la lista fuesen otras listas, `len()` nos devolvería únicamente la cantidad de elementos de la lista principal, sin considerar los elementos de sus *sublistas*.

Por ejemplo:

[25]:
```
lista = [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
print(len(lista))
```

```
4
```

Esto es así, pues para la lista, cada sublista es un *elemento*, no una "extensión" de la lista. En nuestro ejemplo, la lista tiene cuatro sublistas y, por lo tanto, cuatro elementos. Esto también es cierto para cuando los elementos son *strings*, pues cada *string* es un elemento:

[26]:
```
lista = [[1, 2, 3], "Hola", 34]
print(len(lista))
```

```
3
```

1.2.4 4. Cambiar/actualizar un valor

Como vimos previamente, para actualizar o modificar un valor de una lista determinada, se debe acceder a la posición deseada por medio de la indexación y se utiliza el **operador de asignación**:

[27]:
```
lista_pares = [2, 4, 6, 8, 10, 11, 14, 16, 18, 20]
print("Antes de modificar:", lista_pares)
lista_pares[5] = 12
print("Después de  modificar:", lista_pares)
```

```
Antes de modificar: [2, 4, 6, 8, 10, 11, 14, 16, 18, 20]
Después de  modificar: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

1.2.5 5. Eliminar elementos

Es posible eliminar elementos de una lista a través de su posición utilizando el método `.pop()`. Este método recibe una posición, elimina el elemento de dicha posición y lo retorna (en otras palabras, lo "saca"). En caso de que no se indique una posición, el método eliminará el último elemento de la lista. Por ejemplo:

[28]:
```
lista_numeros = [1, 2, 3, 4, "5", 5, 6]
print("Al inicio:", lista_numeros)

# Se desea eliminar el valor "5" que está en formato string, el cual está en la␣
 ↪posición 4
valor = lista_numeros.pop(4)
print("Después de sacar la molestia:", lista_numeros)

# En la variable valor, tendré el valor eliminado
print(valor)
```

```
Al inicio: [1, 2, 3, 4, '5', 5, 6]
Después de sacar la molestia: [1, 2, 3, 4, 5, 6]
5
```

[29]:
```
lista_numeros = [1, 2, 3, 4, 5, 6, 10]

# Quiero eliminar el último elemento
# Puedo usar .pop() sin asignarlo a nada
lista_numeros.pop()
print(lista_numeros)
```

```
[1, 2, 3, 4, 5, 6]
```

Cabe destacar que, si la posición dada es inválida, `pop` generará el error `IndexError`. Por ejemplo:

[30]:
```
lista_numeros = [1, 2, 3, 4, 5, 6, 10]

# Trato de eliminar un elemento que no existe
lista_numeros.pop(12)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[30], line 4
      1 lista_numeros = [1, 2, 3, 4, 5, 6, 10]
      3 # Trato de eliminar un elemento que no existe
----> 4 lista_numeros.pop(12)
```

```
IndexError: pop index out of range
```

1.2.6 6. Cortar (*Slicing*)

Una lista puede ser cortada para extraer parte de su contenido, generando una nueva lista. Para realizar la acción anterior se debe indicar los rangos de datos a copiar.

La forma de cortar una lista es la siguiente:

`lista[posicion_inicial:posicion_final:salto]`

Al igual que la función `range()`, es posible omitir parámetros al utilizar el operador de corte, pero no podemos omitir los dos puntos (`:`), porque de otro modo, lo interpretará como un índice normal.

Cabe destacar que al generar una nueva lista con las posiciones indicadas, la lista original **NO se modifica**, solo se crea una nueva lista (copia).

[31]:
```
lista_numeros = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_cortada = lista_numeros[2:6]
print("Lista original:", lista_numeros)
print("Segmento cortado:", lista_cortada)
```

```
Lista original: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Segmento cortado: [6, 8, 10, 12]
```

Como se observa en el ejemplo, la posición final no es considerada (el final del rango es estrictamente mayor que el último elemento seleccionado), sino que se copia hasta la posición anterior. En este caso particular, Python copiará los datos de la lista que se encuentran en las posiciones 2, 3, 4 y 5. Si en cambio, uso el parámetro de salto:

[40]:
```
lista_numeros = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_cortada = lista_numeros[2:6:2]
print(lista_cortada)
```

```
[6, 10]
```

Solo obtengo las posiciones 2 y 4, pues irá obteniendo elementos de 2 en 2 y 6, el que vendría después del 4, no es menor que 6.

Ahora veamos qué ocurre si omito el parámetro `posicion_inicial`:

[39]:
```
lista_numeros = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_cortada = lista_numeros[:6]
print(lista_cortada)
```

```
[2, 4, 6, 8, 10, 12]
```

En este caso Python, generará una lista desde la primera posición (posición 0) hasta la posición indicada. En el caso particular del ejemplo, se copian los datos de la lista que se encuentran en los índices 0, 1, 2, 3, 4 y 5.

También es posible omitir el parámetro `posicion_final`.

[38]:
```
lista_numeros = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_cortada = lista_numeros[6:]
print(lista_cortada)
```

```
[14, 16, 18, 20]
```

Si no se ingresa una posición final, Python generará una lista desde la posición indicada (6) hasta la posición final. En el caso particular del ejemplo, se copian los datos de la lista que se encuentran en los índices 6, 7, 8 y 9.

[37]:
```
lista_numeros = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_cortada = lista_numeros[::2]
print(lista_cortada)
```

```
[2, 6, 10, 14, 18]
```

Finalmente si no se ingresa una posición inicial ni una posición final, pero si un salto, Python generará una copia de la lista pero con saltos de elementos, omitiendo los índices 1, 3, 5, 7 y 9.

Con el paso, se puede hacer algo más: no necesita ser positivo, sino que puede ser negativo (pero siempre entero):

[42]:
```
lista_numeros = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_cortada = lista_numeros[2:6:-1]
print(lista_cortada)
```

```
[]
```

¿Por qué el resultado es una lista vacía? Pues porque no hay un camino que lleve de 2 a 6, yendo de -1 en -1. Pero si ponemos los índices al revés:

[43]:
```
lista_numeros = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_cortada = lista_numeros[6:2:-1]
print(lista_cortada)
```

```
[14, 12, 10, 8]
```

De 6 a 2 sí se puede ir de -1 en -1, por lo que la lista es entregada en ese orden.

1.2.7 7. Concatenar

Al igual que con los strings, es posible unir los elementos de dos listas usando el operador de **concatenación** (+).

En este caso, la concatenación de dos listas se traduce en colocar los elementos de la `lista_2` después de los elementos de la `lista_1`.

A continuación, se muestra la concatenación de dos listas, almacenando el resultado en una nueva lista (`lista_resultado`).

[44]:
```
lista_1 = [1, 2, 3]
lista_2 = [9, 8, 7]
```

```
lista_final = lista_1 + lista_2
print(lista_final)
```

```
[1, 2, 3, 9, 8, 7]
```

También es posible sobreescribir una lista utilizando el operador de concatenación. En el siguiente ejemplo, se concatenan 3 elementos a la lista_1 y luego se guardan en la misma variable:

[45]:
```
lista_1 = [1, 2, 3]
lista_1 = lista_1 + [9, 8, 7]
print(lista_1)
```

```
[1, 2, 3, 9, 8, 7]
```

Esta última operación es equivalente a utilizar el método `.extend` con la segunda lista como parámetro:

[46]:
```
lista_1 = [1, 2, 3]
lista_1.extend([9, 8 ,7])
print(lista_1)
```

```
[1, 2, 3, 9, 8, 7]
```

El método `.extend`, eso sí, es un poco más versátil: toma como parámetro un **objeto iterable**, es decir, puede tomar una lista o un string u otro objeto que contenga elementos, y lo transformará a lista, para agregarlo al final de la original:

[47]:
```
lista_1 = [1, 2, 3]
lista_1.extend("Patito")
print(lista_1)
```

```
[1, 2, 3, 'P', 'a', 't', 'i', 't', 'o']
```

1.2.8 8. Repetir

Adicionalmente, también es posible utilizar la repetición en listas, utilizando el operador de repetición (`*`). Esta operación repite una cierta cantidad de veces la lista, concatenándola consigo misma.

Cabe señalar que esta operación no modifica la lista inicial.

Por ejemplo:

[48]:
```
lista_numeros = [1, 2, 3, 4]
lista_repetida = lista_numeros*3
print(lista_repetida)
```

```
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

1.2.9 9. Cambiar varios valores

También es posible hacer asignación de varios elementos a la vez. Para ello, se debe seleccionar una parte de la lista, tal como si se estuviera cortando, y a cada elemento de ese sector de la lista se le asigna un nuevo valor, que se extraen de la lista a la derecha de la asignación.

Por ejemplo:

[49]:
```
lista_numerica = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_numerica[2:6] = [1, 1, 1, 1]
print(lista_numerica)
```

```
[2, 4, 1, 1, 1, 1, 14, 16, 18, 20]
```

Vale la pena destacar que al hacer este tipo de asignaciones, es necesario que el tipo de dato que se va a asignar también sea una lista.

Esta lista, sin embargo, no tiene la obligación de ser del mismo largo. Por ejemplo:

[50]:
```
lista_numerica = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
lista_numerica[2:3] = [1, 1, 1, 1]
print(lista_numerica)
```

```
[2, 4, 1, 1, 1, 1, 8, 10, 12, 14, 16, 18, 20]
```

En este caso, agregamos más elementos en la posición donde originalmente solo había como elemento el número 6 (¿por qué solo ese elemento?).

1.3 Más métodos

Debido a la utilidad que tienen las `listas` para ayudar en la resolución de problemas, existe una amplia variedad de funciones y métodos para operar sobre ellas.

Si se desea conocer la totalidad de estos métodos, es posible visitar la documentación de Python respecto a listas, en donde se detallan los otros métodos que tienen estas.

1.4 Iteraciones en Listas

Es común que, a la hora de resolver un problema, tengamos que recorrer las listas utilizando iteración, ya sea para mostrar sus valores, ir modificándolos o para cualquier otro propósito que el problema determine.

Como ya conocemos dos estructuras de control de iteración: `while` y `for-in`, vale la pena mencionar ambas.

1.4.1 Iteración usando: `while`

La manera más común de recorrer una lista es avanzar elemento a elemento por medio de los índices de esta, vale decir, desde el primer elemento, en la posición `[0]`, hasta el último elemento, en la posición `[len(lista) - 1]`.

Para ilustrar esto, consideremos que tenemos una lista de números guardados como `string` y queremos transformar a enteros.

[51]:
```
lista_enteros = ['5', '8', '10', '4', '6', '12', '3']
print(lista_enteros)
```

```
['5', '8', '10', '4', '6', '12', '3']
```

Si quisieramos convertir sus elementos a enteros, no es posible hacer algo como:

```
lista_enteros = int(lista_enteros)
```

Pues arrojaría un `TypeError`:

[52]:
```
lista_enteros = ['5', '8', '10', '4', '6', '12', '3']
lista_enteros = int(lista_enteros)
print(lista_enteros)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[52], line 2
      1 lista_enteros = ['5', '8', '10', '4', '6', '12', '3']
----> 2 lista_enteros = int(lista_enteros)
      3 print(lista_enteros)

TypeError: int() argument must be a string, a bytes-like object or a real number,
 ↪not 'list'
```

¿Por qué ocurre este error? Porque Python transformatá un *valor*, si puede, a un número entero, cosa que no tiene sentido en una lista, pues esta no es transformable a un único número sin ambigüedades. Sin embargo, podríamos usar `while`, para hacerlo *elemento a elemento*.

[53]:
```
lista_enteros = ['5', '8', '10', '4', '6', '12', '3']

# Lista original
print(lista_enteros)

# Desde el primer elemento
i = 0
# Hasta alcanzar el largo de la lista
while i < len(lista_enteros):
  # Convierto elemento a elemento
  lista_enteros[i] = int(lista_enteros[i])
  # Incremento el valor
  i = i + 1

# Imprimo la lista modificada
print(lista_enteros)
```

```
['5', '8', '10', '4', '6', '12', '3']
[5, 8, 10, 4, 6, 12, 3]
```

Si recordamos lo que vimos de for-in, si lo usamos para hacer el mismo procedimiento, este no cambiaría la lista, sino que haría solo el cambio sobre **la variable usada para iterar**. Por ejemplo:

[55]:
```
lista_enteros = ['5', '8', '10', '4', '6', '12', '3']

for e in lista_enteros:
```

```
  e = int(e)

print(lista_enteros)
```

```
['5', '8', '10', '4', '6', '12', '3']
```

Como se puede ver, esto no cambia la lista, porque si bien la variable `e` es transformada en cada iteración, **esta transformación ocurre sobre una copia del valor y no sobre el elemento de la lista**. Si quisiéramos cambiar los valores usando `for-in`, podríamos recorrer por posiciones usando `range()`. Por ejemplo:

[56]:
```
lista_enteros = ['5', '8', '10', '4', '6', '12', '3']

# range(len(lista_enteros)) recorre desde 0 hasta len(lista_enteros) - 1
for i in range(len(lista_enteros)):
  # Al igual que en el caso del while, estoy trabajando por posición
  # NO POR ELEMENTO
  lista_enteros[i] = int(lista_enteros[i])

print(lista_enteros)
```

```
[5, 8, 10, 4, 6, 12, 3]
```

De todos modos, podemos usar el recorrido por valor para casos en los cuáles **no queremos modificar la lista original**. Como por ejemplo, calcular la suma de sus elementos:

[58]:
```
lista_enteros = ['5', '8', '10', '4', '6', '12', '3']

suma = 0
for e in lista_enteros:
  e = int(e)
  suma = suma + e

print(suma)
```

```
48
```

En este caso, como no era de interés para el problema cambiar los elementos de la lista a enteros, transforma en cada paso la variable `e`, para luego acumular el resultado en la variable `suma`.

Importante: Si bien es lícito cambiar los elementos de una lista, como hicimos con `range` antes, lo que *no* debe hacerse es modificar el **tamaño** de la lista que estamos recorriendo (podríamos modificar el de otra, eso sí, por ejemplo, si los números transformados los guardamos en una lista distinta, en lugar de la original). Esto es porque los ciclos `for` llevan un registro interno de en qué posición de la lista van y si la modificamos, este registro generará inconsistencias, como posiciones perdidas o saturación de la memoria del sistema.

1.5 Listas de listas

Como vimos, las listas pueden componerse de cualquier elemento, por lo que es posible (y común) declarar una lista que tenga como elementos en su interior otras listas o **sublistas**. Por ejemplo:

[59]:
```
sub_lista_1 = [1, 0, 1]
sub_lista_2 = [1, 1, 0]
sub_lista_3 = [0, 1, 1]

lista = [sub_lista_1, sub_lista_2, sub_lista_3]
print(lista)
```

```
[[1, 0, 1], [1, 1, 0], [0, 1, 1]]
```

También podemos definirlo todo directamente en una sentencia, aunque hay que poner atención al balanceo de paréntesis y que las líneas no sean demasiado largas (por norma general, una línea de un código debiera tener menos de 80 caracteres de largo).

[60]:
```
lista = [[1, 0, 1], [1, 1, 0], [0, 1, 1]]
print(lista)
```

```
[[1, 0, 1], [1, 1, 0], [0, 1, 1]]
```

Del mismo modo, y especialmente cuando las líneas se volverían demasiado largas, Python acepta saltos de línea, sin cambiar el contenido de la lista:

[61]:
```
lista = [
          [1, 0, 1],
          [1, 1, 0],
          [0, 1, 1]
        ]

print(lista)
```

```
[[1, 0, 1], [1, 1, 0], [0, 1, 1]]
```

Del mismo modo, dependiendo de cuántos niveles tenga la lista, puedo utilizar la sintaxis de indexación varias veces para obtener elementos de las sublistas. Por ejemplo:

[63]:
```
lista_de_listas = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

# Esto obtendría el elemento en la posición 2, que es una sublista,
# y de esta sublista, obtiene la posición 3, es decir, el valor 12
print(lista_de_listas[3][2])

# Y esto obtendría el valor 4, pues está en la sublista 1, en la posición 0
print(lista_de_listas[1][0])
```

```
12
4
```

Esta misma sintaxis se puede usar para modificar los elementos de las sublistas. Por ejemplo:

[ ]:
```
lista_de_listas = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]

# Quiero cambiar el 9 por 9 ** 2
```

```
lista_de_listas[2][2] = lista_de_listas[2][2] ** 2

print(lista_de_listas)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 81], [10, 11, 12]]
```

El entender que las listas pueden componerse de otras y usarlo de forma creativa para resolver problemas es indispensable para generar **abstracciones de datos** complejas, pues con composiciones de listas puedo:

- Representar una matriz numérica.
- Representar un tablero de un juego.
- Representar un mapa.

Si añadimos un nivel más y tuviéramos listas de listas de listas, ya podríamos representar elementos tridimensionales y otras representaciones comunes de datos que encontramos en el día a día, como por ejemplo, las imágenes.

1.5.1 Respecto a las imágenes

Una imagen es un paralelepípedo (lista de lista de listas) de dimensiones $N \times M \times 3$, donde N es el alto, M es el ancho de la imagen y 3 son los valores de **rojo**, **verde** y **azul** de cada pixel.

Normalmente $N \times M$ es la dimensión de la imagen, por ejemplo, 1366×768, y cada "punto" o pixel de esa matríz tiene tres valores de color, donde la intensidad de cada uno se representa normalmente por un valor entre 0 y 255 (esta es una *base*, pero hay otras disponibles, según cuántos bytes se utilicen, que suelen ir de uno a cuatro).

Por ejemplo, la lista `pixels` del ejemplo a continuación, tiene 5 filas con 3 pixeles del mismo color cada una. * La primera fila es de color negro. * La segunda es de color blanco. * La tercera es de color rojo. * La cuarta es de color verde. * La quinta es azul.

En cada una, cada pixel está representado como una lista de tres elementos, con valores entre 0 y 255.

Si bien el código es un poco más avanzado de lo que estamos viendo ahora mismo, es un buen ejemplo de lo que es posible hacer con **listas de listas**, ¡y con pocas líneas de código!

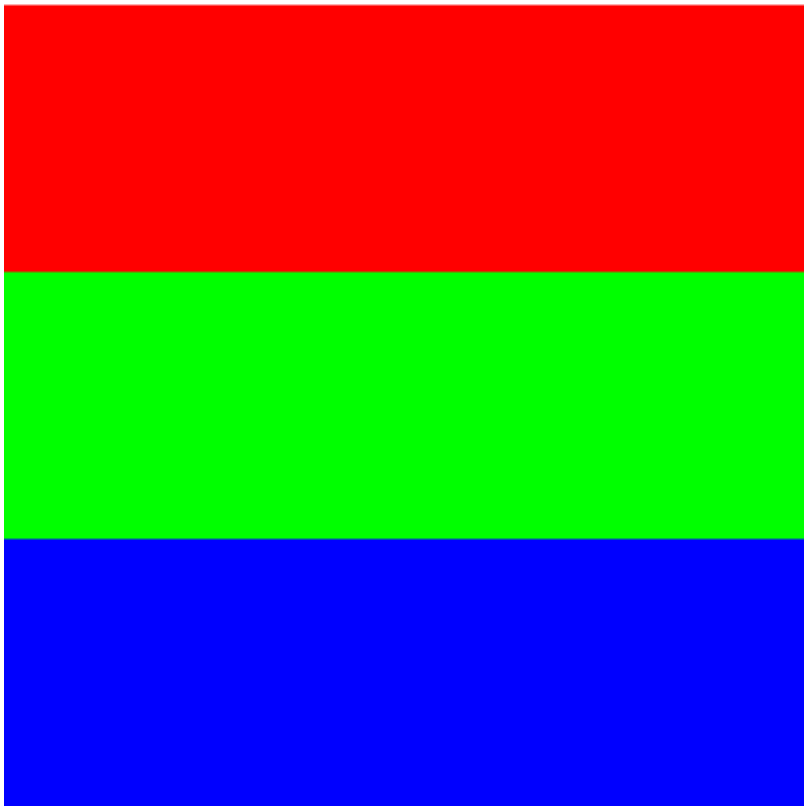
[67]:
```
# Esto es para usar imágenes
from PIL import Image
# Esto es para convertir entre listas de Python y pixeles de PIL
import numpy as np


# Esta es mi imagen de 3 de ancho, 5 de largo con 15 pixeles en total
pixels = [
   [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
   [[255,255,255],[255,255,255],[255,255,255]],
   [[255,0,0],[255,0,0],[255,0,0]],
   [[0, 255, 0], [0, 255, 0], [0, 255, 0]],
   [[0, 0, 255], [0, 0, 255], [0, 0, 255]],
```

```
]

# Convierto la lista a un formato que el módulo PIL entiende
array = np.array(pixels, dtype=np.uint8)

# Creo la imagen
new_image = Image.fromarray(array)
# Como la imagen es de 3x5, es muy pequeña para verla, por lo que la agrandamos
new_image = new_image.resize((300,500), Image.NONE)
# Mostramos la imagen resultante
# new_image.show()
display(new_image)
```



1.6 Listas como referencias

Como ya podemos suponer, una lista puede tener cantidades gigantescas de elementos. Por ejemplo, si almacenaramos un fondo de pantalla de resolución 1920×1080 como una lista en Python, tendríamos 6.220.800 elementos. Es por ello, que a la hora de crear copias sobre listas, estas funcionan de una forma **totalmente distinta a lo que hemos visto hasta ahora**.

Para ilustrarlo, veamos el siguiente ejemplo:

[72]:
```
lista_1 = [1, 2, 3]
lista_2 = lista_1

lista_2[0] = 220

print("Lista original:", lista_1)
print('La lista "copiada"', lista_2)
```

```
Lista original: [220, 2, 3]
La lista "copiada" [220, 2, 3]
```

Al intentar hacer la copia de la `lista_1` en la variable `lista_2` usando el operador asignación, ocurre que cualquier modificación que se haga sobre la `lista_2` **se propaga a la** `lista_1`.

Esto ocurre porque, a diferencia de lo que vimos con los tipos de datos anteriores (`boolean`, `int`, `float`, `complex` y `string`), al usar el operador de asignación sobre una lista, estamos creando solamente una **referencia** (llamadas normalmente "punteros" en otros lenguajes de programación) a la misma, en vez de una copia de los valores.

En términos sencillos, `lista_2` no es una copia nueva de la `lista_1`, sino que es un nuevo nombre para el **mismo elemento**. En este caso, la lista con los elementos `[220, 2, 3]` posee dos nombres, `lista_1` y `lista_2` (es como cuando tenemos un amigo o amiga a quien conocemos por su nombre y un apodo).

Esto se debe a que, como Python no sabe de antemano la cantidad de elementos que podría tener una lista, el hacer una copia de ella podría significar un incremento explosivo del uso de memoria por un programa, además del tiempo que podría tomar copiar los datos de un lado al otro.

Si queremos una copia con los valores, que no sea una referencia a la misma, debemos usar el método `.copy()`. Por ejemplo:

[73]:
```
lista_1 = [1, 2, 3]
lista_2 = lista_1.copy()

lista_2[0] = 220

print("Lista original:", lista_1)
print("Lista copiada:", lista_2)
```

```
Lista original: [1, 2, 3]
Lista copiada: [220, 2, 3]
```

En este caso, podemos ver que al modificar `lista_2`, como ahora es una **nueva lista** y **no una referencia**, sus modificaciones no afectan a la `lista_1`.

1.6.1 Referencias circulares

Otro problema que puede ocurrir por no considerar esta propiedad de las listas, es la creación de referencias circulares. Por ejemplo:

[ ]:
```
lista_1 = [1,2,3]

lista_1[0] = lista_1

print(lista_1)
```

```
[[...], 2, 3]
```

En este caso, el primer elemento de la lista_1 es una **referencia** a si misma. Lo que implica que la lista está conteniéndose a si misma.

Si revisamos que hay en su índice 0, veremos que es exactamente la misma lista:

[ ]:
```
lista_1[0]
```

[ ]:
```
[[...], 2, 3]
```

Si vamos más en profundidad, veremos que el primer elemento de cada sublista, es exactamente la misma lista.

[ ]:
```
lista_1[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]
```

[ ]:
```
[[...], 2, 3]
```

Esto implica que al modificar ese elemento, eliminamos toda esta cadena de referencias. Por ejemplo:

[ ]:
```
lista_1[0][0][0] = 'Mira mami, estoy rompiendo Python'

print(lista_1)
```

```
['Mira mami, estoy rompiendo Python', 2, 3]
```

Si bien, las referencias circulares no son comunes, puede ser que, producto de una mala implementación, nos aparezca una de ellas, en cuyo caso se hace necesario depurar el código para corregir el problema.

1.7 Conversión de string a lista y viceversa

Un problema común con el que nos encontraremos es la conversión de `list` a `string` y hacer la operación inversa.

Para ello, es bueno revisar algunas opciones.

- En primer lugar, vale la pena explorar que hace la función de cambio de tipo: `list()` sobre un string:

[ ]:
```
texto_como_lista = list('Woosh ahora sere una lista')
print(texto_como_lista)
```

```
['W', 'o', 'o', 's', 'h', ' ', 'a', 'h', 'o', 'r', 'a', ' ', 's', 'e', 'r', 'e',
' ', 'u', 'n', 'a', ' ', 'l', 'i', 's', 't', 'a']
```

Como vemos, el problema es que cuando hacemos esta conversión nos genera una lista donde cada caracter es un elemento. Pero a veces, queremos una lista separando por palabra, por línea u otro elemento. En ese caso, es mejor usar el método de los strings `.split(substring)`.

Este método retorna una lista, donde cada elemento es el string separado por el *substring* entregado como parámetro. Por ejemplo:

[75]:
```
texto_como_lista_v1 = 'Woosh ahora sere una lista'.split(' ')
print(texto_como_lista_v1)

texto_como_lista_v2 = 'Woosh ahora sere una lista'.split('o')
print(texto_como_lista_v2)
```

```
['Woosh', 'ahora', 'sere', 'una', 'lista']
['W', '', 'sh ah', 'ra sere una lista']
```

A partir de ambos ejemplos podemos ver que `.split()` usa el caracter como **punto de corte del string**. En el primer caso, ocupamos el caracter espacio `' '` y luego ocupamos el caracter `'o'`. Debemos notar que cuando separamos, lo que usamos para separar no es parte de lo separado, por lo que los perdemos.

También podemos usar un *substring* más largo:

[ ]:
```
texto = 'Hola hola hola hola hola hola hola'.split('la')

print(texto)
```

```
['Ho', ' ho', ' ho', ' ho', ' ho', ' ho', ' ho', '']
```

En este caso usamos el *substring* `'la'` para hacer el corte. Es importante notar que `.split()` solo recibe **un único substring** como caracter de corte. Por lo que si quisiera usar más de uno (Como por ejemplo, separar por espacios y saltos de línea) necesitaría escribir algún procedimiento especial para ello. Del mismo modo, usar `.split()` consecutivos no sirve, pues en el siguiente paso, no tendremos un *string* al cuál aplicarle el método, sino que será una lista. Por lo que al intentarlo nos entregará un error:

[76]:
```
texto = 'Este es un string\nCon espacios y saltos de línea\nTiene 3 líneas'
texto = texto.split('\n')
print(texto)
# La siguiente línea provoca el error
texto = texto.split(' ')
```

```
['Este es un string', 'Con espacios y saltos de línea', 'Tiene 3 líneas']
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[76], line 5
      3 print(texto)
      4 # La siguiente línea provoca el error
----> 5 texto = texto.split(' ')
```

```
AttributeError: 'list' object has no attribute 'split'
```

Un `AttributeError` es un tipo de error que aparece cuando tratamos de usar un método de un objetoal que no corresponde el que estamos usando. En este caso, nos dice que no podemos usar `.split()`, un método de los objetos de la clase *string*, sobre un objeto de la clase `list`.

Hay una forma adicional de usar `.split()`: sin parámetros. En este caso, utiliza un algoritmo diferente para separar: considera que todos los caracteres "en blanco" (espacios, saltos de línea, tabulaciones) son *en su conjunto* el separador. Compara estas dos listas obtenidas a partir del mismo string:

[78]:
```
string = "Tengo muchos    espacios   entre medio y  \
algunas\ttabulaciones\n   o saltos   de línea"

print(string)
# Partimos por espacio
lista_v1 = string.split(" ")
print(lista_v1)

# Partimos sin parámetros, es decir, por "espacios en blanco"
lista_v2 = string.split()
print(lista_v2)
```

```
Tengo muchos    espacios   entre medio y  algunas       tabulaciones
   o saltos   de línea
['Tengo', 'muchos', '', '', '', 'espacios', '', '', 'entre', 'medio', 'y', '',
'algunas\ttabulaciones\n', '', '', 'o', 'saltos', '', '', 'de', 'línea']
['Tengo', 'muchos', 'espacios', 'entre', 'medio', 'y', 'algunas',
'tabulaciones', 'o', 'saltos', 'de', 'línea']
```

Nota los *strings* vacíos en la primera lista: hay lugares con varios espacios juntos y, como entre ellos no hay nada, el elemento separado es un ***string* vacío**.

Por otro lado, si intento hacer el proceso inverso. Es decir, de convertir una lista a un string. También tendremos problemas:

[3]:
```
lista_como_string = str(['Woosh', 'ahora', 'soy', 'una', 'lista'])

print(lista_como_string)
```

```
['Woosh', 'ahora', 'soy', 'una', 'lista']
```

En este caso es más difícil de ver, pero en la variable `lista_como_string` ahora está almacenado el string `'['Woosh', 'ahora', 'soy', 'una', 'lista']'`, con comas, espacios y comillas incluidos. Podemos verlo en el siguiente ciclo, donde se imprime caracter a caracter:

[79]:
```
lista_como_string = str(['Woosh', 'ahora', 'soy', 'una', 'lista'])

for caracter in lista_como_string:
```

```
  # El end=' ' es para que imprima todos los caracteres en la misma línea.
  print(caracter, end=' ')
```

```
[ ' W o o s h ' ,  ' a h o r a ' ,  ' s o y ' ,  ' u n a ' ,  ' l i s t a '
]
```

Del mismo modo que existía el `.split()` en los *string* para transformar a `list`, existe un método de la misma clase que recibe una lista de *strings* y la une en un solo *string* de manera más amigable. Este método es `.join(list)`.

El método une todos los ítems que haya en un elemento iterable (en este caso una lista) y los une usando un separador. La sintaxis de uso es `separador.join(elemento iterable)`. Por ejemplo:

[82]:
```
lista_como_string_v1 = ' '.join(['Woosh', 'ahora', 'soy', 'una', 'lista'])
print(lista_como_string_v1)

lista_como_string_v2 = 'SEP'.join(['Woosh', 'ahora', 'soy', 'una', 'lista'])
print(lista_como_string_v2)

lista_como_string_v3 = ''.join(['Woosh', 'ahora', 'soy', 'una', 'lista'])
print(lista_como_string_v3)
```

```
Woosh ahora soy una lista
WooshSEPahoraSEPsoySEPunaSEPlista
Wooshahorasoyunalista
```

Como se puede observar, el método `.join()` utiliza el *substring* que lo ejecuta como separador y lo usa para concatenar todos los elementos de la lista. En el primer caso, lo hace usando el caracter espacio (`' '`) y en el segundo caso usando el substring `'SEP'`, mientras que en el tercero, la unión la hace un *string* vacío. Es decir, no unimos la lista por un string, sino que **el *string* junta la lista**

Es importante destacar que, para que el método funcione, correctamente **todos los elementos de la lista deben ser de tipo string**. De lo contrario, el programa arrojará un error:

[80]:
```
lista = [1, 2, 3, 4, 5]

#Esta línea genera el error
texto = ' '.join(lista)

print(texto)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[80], line 4
      1 lista = [1, 2, 3, 4, 5]
      3 #Esta línea genera el error
----> 4 texto = ' '.join(lista)
      6 print(texto)
```

```
TypeError: sequence item 0: expected str instance, int found
```

De todos modos, si tenemos este caso, siempre es posible utilizar un ciclo para convertir todos los elementos a *string* para luego usar el método. El construirlo es trivial a estas alturas, pues ya vimos un ejemplo equivalente más arriba, pero se deja una implementación a continuación:

[83]:
```
lista = [1, 2, 3, 4, 5]

# Este while recorre los elementos transformándolos a string.
i = 0
while i < len(lista):
  lista[i] = str(lista[i])
  i = i + 1
# Asi puedo aplicar el .join() sin errores.
texto = ' '.join(lista)

print(texto)
```

```
1 2 3 4 5
```

1.8 Consideraciones finales

A la hora de trabajar con listas, siempre es bueno tener presente el principio de que siempre es más sencillo crear una lista nueva que modificar una existente, por lo que, a veces, cuando en un ejercicio nos pidan alguna modificación sobre una lista, siempre es bueno considerar si vale la pena mover elementos internamente o es mejor crear una copia con la cual trabajar.

Por ejemplo: Si el problema fuese *ordene una lista de menor a mayor*. El algoritmo más sencillo para implementar una solución a este problema consiste en agregar cada elemento ordenadamente a una lista vacía y luego eliminarlo de la lista original (o sea, movemos el menor de la lista original al final de la lista nueva, hasta que no nos quede nada que eliminar, ¡inténtalo!). Las soluciones que mueven elementos dentro de la lista son factibles pero más complejas.

2 Bibliografía

2.1 Listas

Ceder, V. L. (2010). Lists, Tuples and Sets. En *The Quick Python Book* (2.a ed., p. 186). Manning Publications.

GeeksforGeeks. (2022a, julio 21). Python Lists. Recuperado 4 de agosto de 2022, de https://www.geeksforgeeks.org/python-lists/

Python Software Foundation. (2022, 4 agosto). *Data Structures*. Python 3.10.6 documentation. Recuperado 4 de agosto de 2022, de https://docs.python.org/3/tutorial/datastructures.html

2.2 Documentación de Python de `.split()` y `.join()`.

Python Software Foundation. (2023a, julio 02). Built-in Types: Text Sequence Type - str. Recuperado 10 de septiembre de 2023, de https://docs.python.org/3.10/library/stdtypes.html?#text-

sequence-type-str

2.3 Formato y buenas prácticas

Van Rossum, G., Warsaw, B., & Coghlan, N. (2001, 21 julio). *PEP 8 – Style Guide for Python Code.* Python Enhancement Proposals. Recuperado 1 de agosto de 2022, de https://peps.python.org/pep-0008/

2.4 Paso por referencia

Python Software Foundation. (2023) *Programming FAQ. Why did changing list 'y' also change list 'x'?.* Recuperado el 20 de marzo de 2023, de: https://docs.python.org/3/faq/programming.html#why-did-changing-list-y-also-change-list-x