PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

Desarrollo de Software con Python

Cristian Ruz

ÍNDICE

1. ESTRUCTURAS DE DATOS SECUENCIALES

1.1. Listas y sus operaciones

Cuando hablamos de una lista, podemos pensar en una lista de supermercado, en una lista de cosas por hacer, en una lista de personas o participantes de un grupo. A este concepto de ser una **colección** o **agrupación** de datos, le llamamos una **estructura de datos**.

Una lista, entonces va a ser una colección de datos. Pero no cualquier colección, sino que hablaremos de una lista cuando tengamos una colección **ordenada**. Esto significa que cada elemento tiene una **posición definida** dentro de la colección.
En una lista, queremos ser capaces de **acceder a cualquier elemento de ella de manera eficiente**. Si, por ejemplo, queremos saber cuál es el cuarto elemento de la lista queremos ser capaces de encontrarlo rápidamente; si preguntamos cuál es el elemento en la posición 5240, también queremos encontrarlo rápidamente sin tener que mirar los 5239 elementos anteriores. Queremos que encontrar el cuarto elemento sea igualmente rápido que encontrar el elemento en la posición 5240.

1.1.1. La estructura `list`

Python posee un tipo de dato para representar listas: el tipo `list`. Este tipo de dato viene incluído en el intérprete del lenguaje y por lo tanto no es necesario agregar ningún módulo para utilizarlo.

CREANDO UNA LISTA VACÍA Podemos crear una nueva lista vacía invocando la instrucción `list()`:

```
1 lista = list()
```

Podemos mostrar el contenido de una lista utilizando `print`:

```
1 print(lista}
```

```
[]
```

Utilizando la función `type()` podemos comprobar que se trata efectivamente de una estructura de tipo `list`

```
1 print(type(lista))
```

```
<class 'list'>
```

INICIALIZANDO UNA LISTA CON ELEMENTOS También podemos crear una lista entregándole explícitamente los elementos que son parte de ella. Por ejemplo, podemos crear una lista de nombre `profesores` que contiene tres *strings*.

```
1 profesores = ['Jaime', 'Valeria', 'Cristian', 'Carla']
2 print(profesores)
```

```
['Jaime', 'Valeria', 'Cristian', 'Carla']
```

Pero una lista no está limitada a contener solamente elementos de un mismo tipo. Por ejemplo, a continuación definimos una lista llamada `mezcla`,

```
1 mezcla = ['Chile', 8, 4.15, 'Abril']
2 print(mezcla)
```

```
['Chile', 8, 4.15, 'Abril']
```

la cual contiene elementos de tipo `str`, `int` y `float`. Y ya que una lista puede contener elementos de distintos tipos, también pueden incluir otras listas. Por ejemplo, la lista llamada `sublista`,

```
1 sublista = [15, ['Arica', 9], 'Febrero']
2 print(sublista)
```

```
[15, ['Arica', 9], 'Febrero']
```

contiene tres elementos: un `int`, una `list` de dos elementos, y un `str`.

CONSTRUIR LISTA A PARTIR DE OTRAS LISTAS Otra manera de crear listas es a partir de otras listas. Esto lo podemos lograr mediante la **operación de concatenación**, que en Python se representa con el operador `+`. Si tenemos dos o más listas, podemos aplicar la operación de concatenación entre ella, y el resultado es **una nueva lista** que contiene los elementos de la primera, y a continuación los elementos de la segunda.

```
1 otoño = ['Abril', 'Mayo', 'Junio']
2 invierno = ['Julio', 'Agosto']
3 frío = otoño + invierno
4 print(frío)
5 print(otoño)
```

```
['Abril', 'Mayo', 'Junio', 'Julio', 'Agosto']
['Abril', 'Mayo', 'Junio']
```

Luego de la operación de concatenación, la lista concatenada queda almacenada bajo la variable `frío`. Notemos que, a pesar de ello, la variable `otoño` sigue almacenando la misma lista inicial, pues no la hemos modificado. Efectivamente, la operación de concatenación **retorna una nueva lista**.

También podemos utilizar el **operador de multiplicación**, representado por el símbolo `*`, para concatenar una lista consigo misma una cantidad entera de veces, y obtener una nueva lista que repite sus contenidos.

```
1 invierno = ['Julio', 'Agosto']
2 más_frío = invierno * 2
3 print(más_frío)
```

```
['Julio', 'Agosto', 'Julio', 'Agosto']
```

1.1.2. Operaciones sobre `list`

Una vez que tenemos una lista construida, tenemos que saber cómo hacer consultas sobre ella.

LONGITUD DE UNA LISTA En primer lugar, queremos preguntar **cuántos elementos hay en la lista**. La función `len()` nos permite obtener esta información.

```
mezcla = ['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC']
largo = len(mezcla)
print(largo)
```

```
8
```

En este caso, tenemos una lista de 8 elementos.

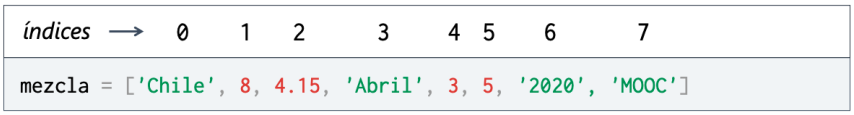

Figura 1.1: Una lista y sus índices. Notar que el primero es 0

BÚSQUEDA POR ÍNDICE Una característica fundamental de las listas es que sus elementos están ordenados. Esto significa que cada uno tiene una posición, que se conoce como índice del elemento en la lista. Usando este índice, podemos consultar entonces, por el elemento que está en una posición específica usando la notación con `[]`. Los índices empiezan siempre en la posición 0, como se puede ver en la figura 1.1.

```
mezcla = ['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC']
print(mezcla[0])
print(mezcla[4])
print(mezcla[7])
print(mezcla[-2])
```

```
Chile
3
MOOC
2020
```

por lo tanto si queremos obtener el primer elemento de la lista, preguntamos por la posición 0. Si queremos obtener el quinto elemento, preguntamos por el elemento en la posición 4, y así hasta que podemos preguntar por el elemento en la posición que corresponde a la longitud de la lista menos uno. En este caso, la posición 7. En el caso particular de Python, podemos utilizar también índices negativos para consultar elementos desde la última posición, hacia la izquierda. Por ejemplo, si preguntamos por el índice -2, obtenemos el elemento en la penúltima posición de la lista.

Esta es una operación que se efectúa de manera **muy eficiente** en una lista, **sin importar la cantidad de elementos** que haya en ella.

EXTRACCIÓN DE ELEMENTOS (*slicing*) Tal como con los *strings*. Las listas permiten usar la notación de *slicing* para extraer porciones de la lista. La notación de *slicing* utiliza los `[]` donde se especifica un índice inicial, y un índice final, separados por `:`. Esto indica que queremos obtener una nueva lista, cuyos elementos van desde el índice inicial hasta la posición justo antes del índice final indicado en la lista original. Por ejemplo, si pedimos la sublista `[2:6]`, obtenemos una nueva lista que contiene las posición 2, 3, 4 y 5 de la lista original.

```
mezcla = ['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC']
print(mezcla[2:6])
print(mezcla[:3])
print(mezcla[5:])
print(mezcla[1:6:2])
```

```
[4.15, 'Abril', 3, 5]
['Chile', 8, 4.15]
[5, '2020', 'MOOC']
[8, 'Abril', 5]
```

Si omitimos el índice inicial, se considera que es 0. Si omitimos el índice final, se considera que es la última posición de la lista. Si agregamos un tercer entero en la notación de *slicing*, esto indica que debemos **saltar elementos** (*step*). Por ejemplo, si pedimos la sublista `[1:6:2]`, entonces significa que queremos los elementos desde la posición 1 hasta la 5 de la lista original, pero avanzando de a 2 posiciones cada vez, por lo tanto obtenemos las posiciones 1, 3, y 5. La notación de *slicing* es muy poderosa, pues nos permite copiar rápidamente una porción de una lista, para obtener una lista nueva.

MODIFICAR ELEMENTOS DE UNA LISTA Podemos modificar elementos individuales de una lista usando la notación de índices. Por ejemplo, podemos modificar los elementos en la posición 3 o en la posición 4, efectuando una asignación donde la notación `mezcla[indice]`, se ubica al lado izquierdo de la asignación.

```
mezcla = ['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC']
mezcla[3] = 'Mayo'
mezcla[4] += 5
print(mezcla)
```

```
['Chile', 8, 4.15, 'Mayo', 8, 5, '2020', 'MOOC']
```

De la misma manera podemos usar la notación de *slicing* para modificar un rango de elementos. Por ejemplo si queremos modificar las posiciones 2, 3 y 4, podemos reemplazarlas por otros valores.

```
mezcla = ['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC']
mezcla[2:5] = [-1, 0, -2]
print(mezcla)
```

```
['Chile', 8, -1, 0, -2, 5, '2020', 'MOOC']
```

Más aún, si intentamos modificar una porción, digamos, de tamaño 5 de la lista, y reemplazarla por una lista de tamaño 1, estaremos reduciendo el tamaño de la lista.

```
mezcla = ['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC']
mezcla[1:6] = ['X']
print(mezcla)
```

```
['Chile', 'X', '2020', 'MOOC']
```

RECORRIENDO LISTAS El hecho que la lista sea una estructura ordenada nos permite utilizar los índices para iterar sobre sus posiciones y acceder de manera ordenada a cada elemento. Una manera directa es usando la instrucción `for i in range(len(lista))`. Esta instrucción nos permite utilizar una variable que avance desde la primera posición de la lista, la posición 0, hasta la última posición de la lista que corresponde al largo de la lista menos uno. En cada iteración, podemos utilizar el índice y la notación de acceso a índices de la lista, con `[]`, para acceder a cada elemento de la lista. Esta manera de recorrer la lista es útil para cuando queremos encontrar un elemento en la lista y además **nos interesa la posición** en que se encuentra ese elemento.

```
mezcla = ['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC']
for i in range(len(mezcla)):
    print(i, "-->", mezcla[i])
```

```
0 --> Chile
1 --> 8
2 --> 4.15
3 --> Abril
4 --> 3
5 --> 5
6 --> 2020
7 --> MOOC
```

Una notación alternativa, y más intuitiva es similar a la que se usa en los `string`. La notación `for elemento in lista` es más sencilla de escribir y nos permite construir un ciclo en que la variable `elemento` toma, en cada iteración, el valor de un elemento de la lista, empezando siempre en el orden desde el primero hasta el último. Escribir la iteración de esta manera es muy conveniente cuando solamente queremos recorrer la lista, por ejemplo, para ver si cada elemento cumple una condición, o para buscar un elemento, pero **no nos importa en qué posición se encuentra cada elemento**.

```
mezcla = ['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC']
for elemento in mezcla:
    print(elemento)
```

```
Chile
8
4.15
Abril
3
```

```
5
2020
MOOC
```

AGREGAR Y SACAR DESDE EL FINAL El tipo de dato `list`, en Python, posee múltiples operaciones y métodos que podemos usar para manipular la lista.[1]. De ellas, las más importantes son las que nos permiten agregar y remover elementos de la lista.
Las listas son, en particular, muy eficientes para agregar y eliminar elementos desde el final. Los métodos correspondientes son append, el cual recibe como argumento un valor, y lo agrega como nuevo elemento **al final de la lista**, de manera que ahora la lista tiene un elemento más.

```
1 opciones = ['Si', 'No']
2 opciones.append('No sé')
3 print(opciones)
```

```
['Si', 'No', 'No sé']
```

El método opuesto es pop, el cual **elimina** y entrega como resultado el elemento que está **en la última posición de la lista**, de manera que ahora la lista tiene un elemento menos.

```
1 cuando = ['Mañana', 'Pasado', 'Hoy']
2 eliminado = cuando.pop()
3 print(cuando)
4 print(eliminado)
```

```
['Mañana', 'Pasado']
Hoy
```

Ambas operaciones se pueden realizar de manera muy eficiente, **sin importar el tamaño que las listas tengan**.

INSERTAR Y ELIMINAR DESDE CUALQUIER POSICIÓN También tenemos operaciones para **insertar elementos en cualquier posición**, y para **eliminar elementos**.
El método `insert` recibe una posición, y un elemento, e **ingresa el elemento en la posición indicada** de la lista, **desplazando** todos los que están a continuación, una posición a la derecha.

```
1 opciones = ['Si', 'No', 'No sé']
2 opciones.insert(2, 'Quizás')
3 print(opciones)
```

```
['Si', 'No', 'Quizás', 'No sé']
```

El método pop, por otro lado puede recibir **opcionalmente** un argumento que es la **posición que queremos eliminar**. En este caso, pop **elimina y retorna** el elemento que se encontraba en la posición indicada, desplazando todos los elementos siguientes una posición a la izquierda.

[1] Podemos consultar todas en
https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

```python
cuando = ['Mañana', 'Pasado', 'Hoy']
eliminado = cuando.pop(1)
print(cuando)
print(eliminado)
```

```
['Mañana', 'Hoy']
Pasado
```

ENCONTRAR Y ELIMINAR UN ELEMENTO Las siguientes operaciones trabajan con elementos existentes en la lista.

El método `index` permite **retornar la posición** de un elemento que se encuentra en la lista. En caso que el elemento se encuentre repetido en la lista, retorna **la primera posición** en que lo encuentra. Este método supone que el elemento se encuentra en la lista. Si no es así, se produce un error, por lo que debemos estar seguros que el elemento existe antes de preguntar por su índice.

```python
opciones = ['Si', 'No', 'Quizás', 'No sé']
print(opciones.index('Quizás'))
print(opciones)
```

```
2
```

El método `remove` recibe un elemento y **lo elimina de la lista**. Si el elemento se encuentra repetido en la lista, **elimina solamente la primera ocurrencia**. Al igual que `index`, este método supone que el elemento ya existe en la lista, de lo contrario se produce un error. La ejecución de **remove modifica la lista**, y **no entrega un valor de retorno**.

```python
cuando = ['Mañana', 'Pasado', 'Hoy']
cuando.remove('Pasado')
print(cuando)
```

```
['Mañana', 'Hoy']
```

1.2. Aplicaciones de listas

Si bien, en una lista podemos almacenar cualquier tipo de elementos, y mezclar elementos de distintos tipos, en la práctica es muy común que utilicemos listas para guardar elementos que sean todos del mismo tipo. Por ejemplo, una lista de estudiantes que guarda *strings*, una lista de puntajes, que guarda enteros, una lista de respuestas que guarda *strings*, o una lista de temperaturas que guarda `float`s.

Recordemos que una de las características de las listas es que los elementos están ordenados, por lo tanto esta estructura es adecuada cuando la posición en que aparecen los elementos en la lista es importante. El hecho que estén ordenados, además nos permite recorrer sistemáticamente todos los elementos de una lista, conocer su posición, y hacer algún procesamiento con ellos.

Tomando como ejemplo las siguientes listas:

```
estudiantes = ['Aurora', 'Sebastián', 'Rafaela', 'Dario', 'Lisa',
 ↪  'Almendra', 'Camilo']
puntos = [67, 56, 48, 21, 60, 38, 89, 20, 28, 79]
temperaturas = [16.8, 35.2, 27.9, 19.2, 19.8, 24.8, 34.3, 19.0, 37.3,
 ↪  20.6]
respuestas = ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']
```

Algunas preguntas que podemos querer hacer con estas listas podrían ser:

- ¿cuál es el o los nombres más largos?
- ¿cuál es el promedio de puntos?
- ¿hay dos respuestas iguales seguidas?
- ¿en qué posición ocurre la temperatura más alta?

1.2.1. Ejemplos de uso de listas

BUSCAR NOMBRE MÁS LARGO Supongamos que tenemos una lista de strings con nombres de estudiantes:

```
estudiantes = ['Aurora', 'Sebastián', 'Rafaela', 'Dario', 'Lisa',
 ↪  'Almendra', 'Camilo']
```

y queremos encontrar aquél que tiene el nombre más largo. En este caso vamos a querer aplicar la función `len()` a cada elemento, para obtener su longitud y luego seleccionar la de mayor valor. Podemos pensar que una opción sería re-ordenar los *strings* de acuerdo a su longitud, para quedarnos con el primero. Esta es una idea que funciona, pero vamos a hacerla menos atractiva al pedir como respuesta no solo el *string* más largo, sino también la posición (índice) en que se encuentra. Con esta requisito, el reordenar la lista nos hace perder un poco de información, y nos exigiría tener que mantener una copia de la lista original para no perder las posiciones de cada *string*, con lo cual también estaríamos duplicando el uso de memoria; no parece ser mucho problema para esta lista, pero si fuera en una lista de un millón de elementos, ya no es tan buena idea.

Aprovecharemos el hecho que los elementos están ordenados para plantear una solución más sencilla.

En primer lugar vamos a utilizar dos variables: una para almacenar la posición del *string* más largo, que, supondremos, será inicialmente el primero (índice 0); la otra variable almacenará el *string* más largo visto hasta el momento, el que inicializaremos con el primer elemento de la lista. A continuación, utilizaremos el recorrido con `for i in range(len(lista))` para recorrer todas las posiciones.

```
pos_mas_largo = 0
mas_largo = estudiantes[0]

for i in range(1,len(estudiantes)):
    if len(estudiantes[i]) >= len(mas_largo):
        pos_mas_largo = i
        mas_largo = estudiantes[i]

print(f"El nombre más largo es {mas_largo} y se encuentra en la posición
 ↪  {pos_mas_largo}")
```

```
El nombre más largo es Sebastián y se encuentra en la posición 1
```

Fijémonos en que este ciclo **itera a través de las posiciones**, y no de los elementos, porque nos interesa saber en qué posición se encuentra el elemento que buscamos. Dentro de cada iteración nos preguntamos, con un `if`, si el elemento que estamos mirando es más largo que el más largo que hemos visto hasta el momento y, de ser así, actualizamos primero la posición del elemento más largo, y luego actualizamos el elemento más largo visto hasta el momento. De esta manera, una vez que la iteración termina, podemos entregar la respuesta.

Esta solución tiene una particularidad: si hay más de un elemento que es el más largo, nos entrega como resultado el último que aparece en la lista. Si queremos que nos entregue el primero, solamente debemos cambiar la condición `>=`, por `>`, de manera que la actualización se produzca solo cuando haya un elemento efectivamente más largo que el actual.

Finalmente, ¿qué podemos hacer si queremos obtener todos los elementos que tengan la máximo longitud, pero sin tener que recorrer la lista una segunda vez? Como pista, tendremos que usar listas también en nuestras variables `pos_mas_largo`, y `mas_largo`.

CALCULAR PROMEDIO DE ELEMENTOS Veamos ahora el caso en que tenemos una lista de puntajes y queremos obtener el promedio de estos puntajes.

```
1 puntos = [67, 56, 48, 21, 60, 38, 89, 20, 28, 79]
```

Esta vez no nos interesa la posición, por lo tanto podemos utilizar la iteración usando la construcción `for elemento in lista` y, en cada paso de la iteración, acumular la suma en una variable de nombre `suma` que inicializamos en 0. Una vez terminada la iteración, dividimos por la cantidad de puntos en la lista, utilizando la función `len`, y así obtenemos nuestra respuesta.

```
1 suma = 0
2 for elem in puntos:
3     suma += elem
4 promedio = suma/len(puntos)
5 print(f"El promedio de puntos es {promedio}")
```

```
El promedio de puntos es 50.6
```

El tipo `list` posee también una función llamada `sum` que se puede aplicar a una lista y retorna la suma de sus elementos. Nuestro problema entonces también podría resolverse en una línea dividiendo el resultado de `sum` por el resultado de `len`. Podríamos pensar entonces, ¿para qué usar la versión larga en que tenemos que escribir la iteración?

```
1 promedio = sum(puntos)/len(puntos)
2 print(f"El promedio de puntos es {promedio}")
```

```
El promedio de puntos es 50.6
```

Hagamos una vez más que la solución simple sea menos atractiva. Pidamos **el promedio de todos los puntajes mayores a 60**. En este caso la solución anterior no nos sirve, y tenemos que escribir el ciclo, al cual le podemos agregar una condición extra con `if` para que solo sume

aquellos elementos que cumplen la condición. Pero no solo eso, tampoco podemos usar `len` tal como antes, ya que solo tenemos que considerar la cantidad de elementos que son mayores a 60.

```
limite_inferior = 60
suma = 0
num_elementos = 0
for elem in puntos:
    if elem >= limite_inferior:
        suma += elem
        num_elementos += 1
promedio = suma / num_elementos
print(f"El promedio de puntos mayores a {limite_inferior} es \
{promedio}")
```

```
El promedio de puntos mayores a 60 es 73.75
```

Se pueden hacer más preguntas respecto a este código: ¿Qué ocurre si no hay elementos mayores a 60? En ese caso este programa dejaría de funcionar pues si `num_elementos` es 0, se produciría un error. ¿Cómo arreglarlo? Y otro desafío: ¿qué pasa si queremos conocer el elemento más cercano al promedio?

EXTRAER ELEMENTOS QUE CUMPLEN UNA CONDICIÓN También podemos resolver problemas en que la respuesta sea una lista que vamos construyendo. Supongamos que tenemos una lista de temperaturas de varios días, donde por supuesto, el orden de los días es importante. Podemos preguntarnos en qué días ha habido temperaturas mayores a, digamos, 30 grados, y cuáles han sido.

```
temperaturas = [16.8, 35.2, 27.9, 19.2, 19.8, 24.8, 34.3, 19.0, 37.3,
 ↪  20.6]
```

En este caso usaremos una variable `limite` para establecer ese valor umbral. Es una buena práctica usar variables para estos límites, ya que si más adelante queremos cambiar este límite a, digamos 25 o 33, solo debemos cambiar la variable `limite`, y no todas las veces que escribimos 30 en nuestro código.

Ya que nos interesa conocer las posiciones, que indican los días de la secuencias, usamos el recorrido `for i in range(len(lista))`. En cada iteración verificamos si la temperatura cumple la condición, y de ser así, construimos dos listas usando `append`: una lista con las temperaturas mayores a 30, y una lista con los días en que se produjeron estas temperaturas. De esta manera, la respuesta a la pregunta son dos listas.

```
limite = 30
dias_altas = []
altas = []
for i in range(len(temperaturas)):
    if temperaturas[i] >= limite:
        dias_altas.append(i)
        altas.append(temperaturas[i])
print(f"Los días {dias_altas} hubo temperaturas mayores a {limite} y\
 fueron {altas}")
```

```
Los días [1, 6, 8] hubo temperaturas mayores a 30 y fueron [35.2, 34.3,
 ↪  37.3]
```

BUSCAR UN PATRÓN EN UNA PORCIÓN DE LA LISTA Supongamos que tenemos una lista de *strings* que representa la secuencia de respuestas a una serie de preguntas de alternativas. Queremos saber si existe una secuencia donde haya **dos respuestas iguales consecutivas**.

```
respuestas = ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']
```

En este caso vamos a usar el recorrido `for i in range(len(lista))`, pero con el cuidado de queremos recorrer todas las sublistas de elementos consecutivos de largo 2. Por lo tanto nuestro `range` no debe llegar hasta el final de la lista, sino hasta la posición anterior al final. ¿Por qué?, porque en cada iteración vamos a comparar la posición `i` con la posición siguiente, `i+1`, de manera que nuestra variable no puede llegar a ser la úlitma posición. De lo contrario, si estamos en la última posición y accedemos a la posición `i+1`, obtendríamos un error.

```
dos_iguales = []
for i in range(len(respuestas)-1):
    if respuestas[i] == respuestas[i+1]:
        dos_iguales = respuestas[i:i+2]
print(f"Respuestas consecutivas iguales: {dos_iguales}")
```

```
Respuestas consecutivas iguales: ['d', 'd']
```

Ya que lo queremos guardar son las evidencias de que hay dos respuestas iguales consecutivas, vamos a guardar en nuestro resultado, que está en la variable de nombre `dos_iguales`, una lista con las dos respuestas iguales consecutivas. Usamos aquí la notación de *slicing* para copiar desde la posición `i` hasta antes de la posición `i+2` (copiando entonces dos elementos).

Así obtenemos nuestra respuesta. Ahora, para hacerlo más desafiante, ¿qué ocurre si queremos todas las secuencias consecutivas iguales y su posición? En este caso, tenemos que usar una lista para guardar las secuencias de respuestas iguales consecutivas, y otra lista para guardar sus posiciones.

```
pos_iguales = []
dos_iguales = []
for i in range(len(respuestas)-1):
    if respuestas[i] == respuestas[i+1]:
        dos_iguales.append(respuestas[i:i+2])
        pos_iguales.append([i,i+1])
print(f"Respuestas consecutivas iguales: {dos_iguales}, en las \
 ↪  posiciones: {pos_iguales}")
```

```
Respuestas consecutivas iguales: [['c', 'c'], ['d', 'd']], en las
 ↪  posiciones: [[1, 2], [4, 5]]
```

Ahora durante la iteración, utilizamos `append` para guardar cada una de las secuencias que cumplen la condición. Y además, en la lista `pos_iguales`, guardamos una lista de dos elementos con las posiciones. De esta manera, el resultado son dos listas: una lista contiene listas de

respuestas iguales consecutivas, y la otra lista contiene listas de posiciones donde están esas respuestas.
¿Cómo podríamos extender este si queremos n respuestas iguales consecutivas?

1.2.2. Recomendaciones de uso de listas

NO MODIFICAR UNA LISTA MIENTRAS LA RECORREMOS Tomemos el ejemplo de la lista de respuestas.

```
respuestas = ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']
```

Supongamos que queremos eliminar todas las respuestas que son 'c'. Las listas poseen un método `remove(e)` que elimina la primera ocurrencia de `e` en la lista. Por lo tanto, vamos a recorrer la lista por cada elemento y, si este coincide con 'c', entonces lo eliminamos.

```
print(f"Respuestas: {respuestas}")
for elem in respuestas:
    if elem == 'c':
        respuestas.remove(elem)
print(f"Respuestas: {respuestas}")
```

```
Respuestas: ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']
Respuestas: ['a', 'b', 'd', 'd', 'a', 'c', 'e', 'b']
```

Pero, ¿qué pasa? Se borraron dos 'c', pero la tercera no.

Tal vez nuestro problema es el método `remove`. Usemos `pop`, que es lo que ya conocemos. Aquí tenemos que usar el recorrido por índice con `for i in range(len(lista))`. Cada vez que vemos una 'c', eliminamos esa posición con `pop(i)`.

```
print(f"Respuestas: {respuestas}")
for i in range(len(respuestas)):
    if respuestas[i] == 'c':
        respuestas.pop(i)
print(f"Respuestas: {respuestas}")
```

```
Respuestas: ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']
Traceback (most recent call last):
... ...
IndexError: list index out of range
```

Sin embargo, al ejecutar, esto provoca un error de tipo `IndexError`, que indica que hemos intentado acceder a un elemento fuera del rango de la lista. ¿Por qué ocurre esto? Al iniciar un ciclo que itera sobre una lista, Python prepara el recorrido para la lista original (que tiene todos los elementos) Cada vez que agregamos o sacamos elementos, estamos cambiando la estructura de la lista que ya empezamos a recorrer, y en particular su cantidad de elementos. Esto confunde el ciclo original, pues empezamos a recorrer una lista de 10 elementos, y de un moemento a otro tenemos una de 8. La moraleja de esto es que **nunca debemos cambiar la cantidad de elementos de una lista mientras la estamos recorriendo**, ya que podemos encontrar comportamientos inesperados.

¿Cómo debemos hacerlo, entonces? Cuando queremos eliminar elementos usando un ciclo, debemos **crear una nueva lista sin los elementos que nos interesan**. Entonces creamos una lista vacía de nombre `respuesta_sin_c`, y en ella vamos agregando todos los elementos, excepto aquellos que son una `'c'`.

```
respuestas_sin_c = []
print(f"Respuestas: {respuestas}")
for elem in respuestas:
    if elem != 'c':
        respuestas_sin_c.append(elem)
print(f"Respuestas: {respuestas_sin_c}")
```

```
Respuestas: ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']
Respuestas: ['a', 'b', 'd', 'd', 'a', 'e', 'b']
```

COPIAR LISTAS Otra situación con la que hay tener cuidado al usar listas es al momento de hacer copias de ellas.
Estamos acostumbrados a un comportamiento como el siguiente:

```
a = 42
b = a
b = b+1
print(f"a={a}")
print(f"b={b}")
```

```
a=42
b=43
```

cuando escribimos `b=a`, `b` está asociado al mismo valor que `a`. Si a `b` se le asigna otro valor, entonces `b` queda asociado a un valor distinto, pero el valor asociado a `a` no cambia. `b` simplemente ha cambiado su asociación, como se ve en la figura 1.2

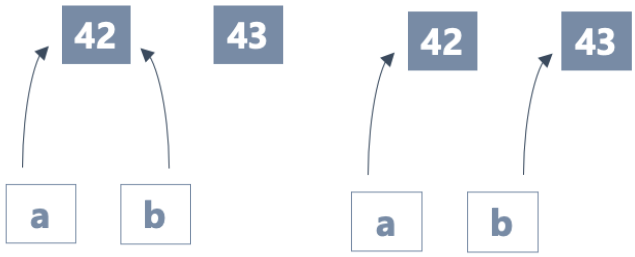

Figura 1.2: Las variables `a` y `b` antes y después de `b=b+1`

Con las listas pasa algo que puede parecer poco intuitivo, pero en el fondo es la misma idea.

```
puntos = [67, 56, 48, 21, 60]
copia = puntos
copia[3] = 0
print(f"Puntos: {puntos}")
print(f"Copia : {copia}")
```

```
Puntos: [67, 56, 48, 0, 60]
Copia : [67, 56, 48, 0, 60]
```

Cuando asignamos una lista a una variable, la variable queda asociada a la lista, **pero no a sus valores**. Esto ocurre porque la lista es solamente una estructura que **contiene asociaciones a valores**. Por lo tanto al asignar `copia=puntos`, la variable `copia` queda asociada a la misma lista que `puntos`. Por lo tanto cuando se modifica la posición 3 de `copia` también se está modificando la posición 3 de `puntos`. La figura 1.3 ilustra esta situación.

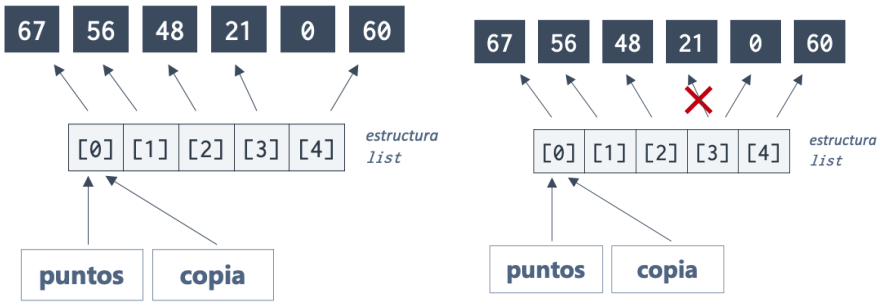

Figura 1.3: Las listas `puntos` y `copia` antes y después de `copia[3] = 0`

Entonces ¿como obtener copias de listas? Hay al menos 3 maneras distintas. Podemos usar el *slicing* de listas. El *slicing* siempre genera listas nuevas, por lo tanto si pedimos un *slice* de la lista completa, lo cual indicamos escribiendo `lista[:]`, obtenemos **una copia completa de la lista**. Obtener una copia, significa que tenemos otra estructura de tipo `list`, distinta a la original. cuya posiciones están asociadas a los mismos valores que la lista original. Por lo tanto, si modificamos el valor asociado a la posición 3 de la lista `copia`, no estamos cambiando la asociación de la posición 3 de la lista `puntos`, ya que se trata de estructuras `list` distintas. La figura 1.4 ilustra el comportamiento de las referencias.

```
1 puntos = [67, 56, 48, 21, 60]
2 copia = puntos[:]
3 copia[3] = 0
4 print(f"Puntos: {puntos}")
5 print(f"Copia : {copia}")
```

```
Puntos: [67, 56, 48, 21, 60]
Copia : [67, 56, 48, 0, 60]
```

Una segunda manera es utilizando el inicializador de listas, `list()`, a partir de la lista original. Esta genera una copia de la lista que recibe, y se produce la misma situación que en la figura 1.4.

```
1 puntos = [67, 56, 48, 21, 60]
2 copia = list(puntos)
3 copia[3] = 0
4 print(f"Puntos: {puntos}")
5 print(f"Copia : {copia}")
```

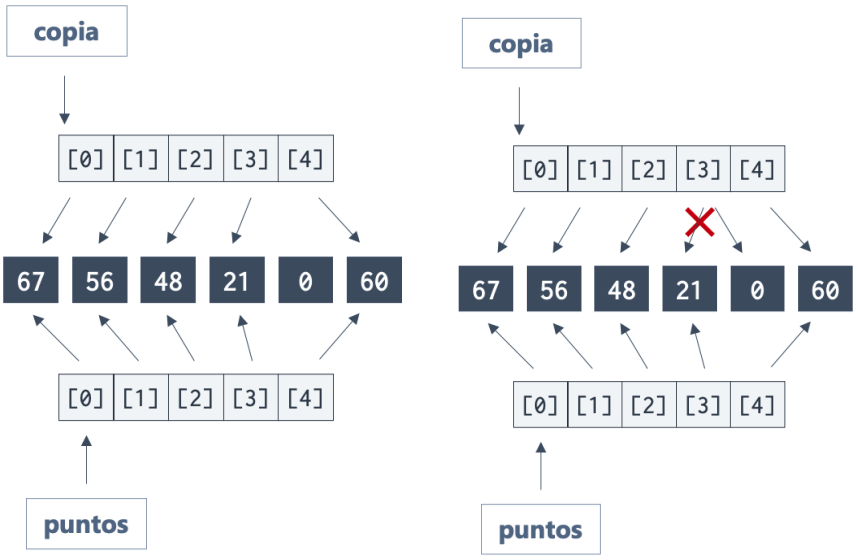

Figura 1.4: Las listas `puntos` y `copia` antes y después de `copia[3] = 0`

```
Puntos: [67, 56, 48, 21, 60]
Copia : [67, 56, 48, 0, 60]
```

Y una tercera manera es usar el método `copy()` de la estructura `list`, el cual entrega como resultado una lista nueva con los elementos copiados de la original. Una vez más, el comportamiento es equivalente al indicado en la figura 1.4, y se obtiene una copia independiente de la lista original.

```
1 puntos = [67, 56, 48, 21, 60]
2 copia = puntos.copy()
3 copia[3] = 0
4 print(f"Puntos: {puntos}")
5 print(f"Copia : {copia}")
```

```
Puntos: [67, 56, 48, 21, 60]
Copia : [67, 56, 48, 0, 60]
```

1.2.3. Ordenar listas

Una acción bastante frecuente en lista es **ordenar sus elementos** de acuerdo a algún criterio.

Python posee una función `sorted()` la cual recibe una lista y entrega **una copia ordenada de la lista**. Como es una copia, la lista original se mantiene sin modificaciones.

```
1 puntos = [67, 56, 48, 21, 60, 38, 89, 20, 28, 79]
2 ordenada = sorted(puntos)
3 print(f"Puntos:   {puntos}")
4 print(f"Ordenada: {ordenada}")
```

```
Puntos:   [67, 56, 48, 21, 60, 38, 89, 20, 28, 79]
Ordenada: [20, 21, 28, 38, 48, 56, 60, 67, 79, 89]
```

Por otro lado, las listas poseen un método de nombre `sort`, el cual se puede aplicar a una lista y **modifica la lista** de manera de ordenar sus elementos de menor a mayor. Este método **NO RETORNA una copia de la lista**, si no que modifica los elementos que están dentro de ellos para que quedan ordenados de menor a mayor.

```
1 puntos = [67, 56, 48, 21, 60, 38, 89, 20, 28, 79]
2 puntos.sort()
3 print(f"Puntos:   {puntos}")
```

```
Puntos:   [20, 21, 28, 38, 48, 56, 60, 67, 79, 89]
```

Si queremos que ordene la lista de manera inversa, podemos entregar un parámetro booleano de nombre reverse.

```
1 puntos = [67, 56, 48, 21, 60, 38, 89, 20, 28, 79]
2 puntos.sort(reverse=True)
3 print(f"Puntos:   {puntos}")
```

```
Puntos:   [89, 79, 67, 60, 56, 48, 38, 28, 21, 20]
```

1.3. Tuplas y sus operaciones

Una tupla, antes que todo, **es una estructura de datos**. Una colección de datos que permite agruparlos y provee operaciones para accederlos.
Una tupla permite agrupar datos de distintos tipos, de manera ordenada. Esto significa que cada elemento tiene una posición definida dentro de la estructura. Además permite acceder de manera eficiente a cada elemento a partir de su posición.
¿Suena conocido? Todo lo que hemos dicho hasta ahora suena igual que una lista. De hecho, el concepto de tupla no es distinto que el de una lista: elementos agrupados, ordenados y con acceso eficiente a cada posición. Pero, si bien en su concepto no hay diferencias, en su implementación en Python sí las hay.

1.3.1. La estructura `tuple`

El tipo de dato `tuple` permite representar tuplas. La inicialización de tuplas se hace de manera similar a como lo hacíamos con las listas.

CREANDO UNA TUPLA VACÍA Podemos obtener una nueva tupla vacía invocando la instrucción `tuple()`.

```
1 tupla = tuple()
```

Utilizando `print` podemos ver el contenido que, en este caso, sería una tupla vacía.

```
1 print(tupla)
```

```
()
```

Las tuplas se delimitan en Python usando paréntesis `()`, a diferencia de las listas, que usan `[]`. Mediante la función `type`, podemos comprobar que se trata efectivamente de una estructura de tipo 'tuple'

```
print(type(tupla))
```

```
<class 'tuple'>
```

INICIALIZANDO UNA TUPLA CON ELEMENTOS Tal como las listas, podemos crear una tupla entregando explícitamente los elementos que son parte de ella, separados por comas, y rodeados por `()`. Por ejemplo, podemos crear una tupla de nombres de profesores que contiene tres *strings*.

```
profesores = ('Jaime', 'Valeria', 'Cristian', 'Carla')
print(profesores)
```

```
 ('Jaime', 'Valeria', 'Cristian', 'Carla')
```

Una tupla tampoco está limitada a contener solamente elementos de un mismo tipo. Por ejemplo, la tupla llamada `mezcla`, contiene elementos de tipo `str`, `int` y `float`.

```
mezcla = ('Chile', 8, 4.15, 'Abril')
print(mezcla)
```

```
('Chile', 8, 4.15, 'Abril')
```

Y así como también puede contener elementos de distintos tipos, esto incluye también listas u otras tuplas. Por ejemplo, la tupla llamaba `sublista`, contiene tres elementos: un `int`, una lista de dos elementos, y un `str`; y la tupla llamada `subtupla` contiene a otra tupla. Podemos ver que la notación con `[]` o `()` nos indica si se trata de una lista o de una tupla.

```
sublista = (15, ['Arica', 9], 'Febrero')
subtupla = (15, ('Arica', 9), 'Febrero')
print(sublista)
print(subtupla)
```

```
(15, ['Arica', 9], 'Febrero')
(15, ('Arica', 9), 'Febrero')
```

CONSTRUIR UNA TUPLA A PARTIR DE OTRAS TUPLAS Otra manera de crear tuplas es mediante concatenación. Si tenemos dos o más tuplas, podemos aplicar la operación de concatenación usando el operador `+`. El resultado de esta operación es una nueva tupla que contiene los elementos de la primera, y a continuación los elementos de la segunda.

```
otoño = ('Abril', 'Mayo', 'Junio')
invierno = ('Julio', 'Agosto')
frío = otoño + invierno
print(frío)
```

```
('Abril', 'Mayo', 'Junio', 'Julio', 'Agosto')
```

De la misma manera podemos utilizar el operador $*$ para concatenar una tupla consigo misma una cantidad entera de veces, y obtener una nueva tupla que repita sus contenidos.

```
otoño = ('Abril', 'Mayo', 'Junio')
invierno = ('Julio', 'Agosto')
más_frío = invierno * 3
print(más_frío)
```

```
('Julio', 'Agosto', 'Julio', 'Agosto', 'Julio', 'Agosto')
```

1.3.2. Operaciones sobre `tuple`

Una vez que tenemos una tupla construida, también podemos hacer consultas sobre ella.

LONGITUD DE UNA TUPLA Para saber cuántos elementos hay en la tupla, utilizamos la función `len()`. En este caso, tenemos una tupla de 8 elementos.

```
mezcla = ('Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC')
largo = len(mezcla)
print(largo)
```

```
8
```

BÚSQUEDA POR ÍNDICE Aprovechando el hecho que los elementos de las tuplas se encuentran ordenados y cada uno de ellos tiene una posición definida, a la cual llamamos **índice del elemento en la tupla**, podemos consultar por cada posición de manera específica.

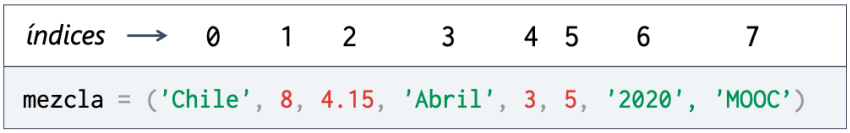

Figura 1.5: Una tupla y sus índices. Notar que el primero es 0

Recordando que los índices empiezan siempre desde 0, como se ven la figura 1.5, podemos preguntar por el primer elemento, por el quinto, por el octavo o por el penúltimo usando estos ejemplos. La notación para indicar un índice sigue siendo indicar el índice entre `[]`. Esta operación, en particular es **muy eficiente** en una tupla sin importar la cantidad de elementos que haya en ella.

```
mezcla = ('Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC')
print(mezcla[0])
print(mezcla[4])
print(mezcla[7])
print(mezcla[-2])
```

```
Chile
3
MOOC
2020
```

EXTRACCIÓN DE ELEMENTOS (*slicing*) Las tuplas también permiten usar la notación de *slicing* para extraer porciones de la tupla. La notación de *slicing* utiliza `[]` donde se especifica un índice inicial, y un índice final, separados por `:`. Esto indica que queremos obtener **una nueva tupla** (una copia), cuyos elementos van desde el índice inicial hasta la posición justo antes del índice final indicado en la tupla original. Por ejemplo, si pedimos la subtupla `[2:6]`, obtenemos una nueva tupla que contiene las posiciones 2, 3, 4 y 5 de la tupla original.

```
mezcla = ('Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC')
print(mezcla[2:6])
print(mezcla[:3])
print(mezcla[5:])
print(mezcla[1:6:2])
```

```
(4.15, 'Abril', 3, 5)
('Chile', 8, 4.15)
(5, '2020', 'MOOC')
(8, 'Abril', 5)
```

Si omitimos el índice inicial, se considera que es 0. Si omitimos el índice final, se considera que es la última posición de la tupla. Si agregamos un tercer entero en la notación de *slicing*, esto indica que debemos saltar elementos. Por ejemplo, si pedimos la subtupla `[1:6:2]`, entonces significa que queremos los elementos desde la posición 1 hasta la 5 de la tupla original, pero avanzando de a dos posiciones cada vez, por lo tanto obtenemos las posiciones 1, 3, y 5. La notación de *slicing* es muy poderosa, pues nos permite copiar rápidamente una porción de una tupla, para obtener una tupla nueva.

MODIFICAR ELEMENTOS DE UNA TUPLA La siguiente operación que quisiéramos ver es cómo modificar elementos de una tupla.

```
mezcla = ('Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC')
mezcla[3] = 'Mayo'
print(mezcla)
```

```
TypeError: 'tuple' object does not support item assignment
```

Sin embargo aquí nos encontramos con la diferencia más importante entre una tupla y una lista en Python. Al intentar asignar un valor a una posición de una tupla ya creada, obtenemos un error del tipo `TypeError`, que indica que **las tuplas no permiten asignar valores a sus elementos**.
Esto significa que no podemos modificar los elementos de una tupla una vez que ya la hemos creado. Esta es la principal diferencia entre una tupla y una lista. Diremos que las tuplas son estructuras **inmutables**, mientras que las listas, donde sí podemos modificar cada elemento de manera individual, son estructuras **mutables**.
Esta característica de **inmutabilidad** la comparten tanto las tuplas como los `str`, y tiene implicancias en otras estructuras.

RECORRIENDO TUPLAS Las tuplas, al igual que las listas, son estructuras **iterables**. Esto significa que las podemos usar dentro de instrucciones como `for i in range(len(tupla))`, y dentro de `for elemento in tupla`, para recorrer cada uno de sus elementos. La diferencia entre ambas maneras es que al utilizar la construcción `for i in range`, controlamos el acceso a cada posición, mediante la variable índice que utilizamos para la iteración. Utilizaremos esta manera cuando queramos recorrer la tupla y hacer algo con la posición de cada elemento.

```
1 for i in range(len(mezcla)):
2     print(i, "-->", mezcla[i])
```

```
0 --> Chile
1 --> 8
2 --> 4.15
3 --> Abril
4 --> 3
5 --> 5
6 --> 2020
7 --> MOOC
```

Por otro lado, la construcción `for elemento in tupla`, nos permite recorrer de manera sistemática la tupla, y acceder a cada elemento directamente usando la variable de la iteración.

```
1 for elemento in mezcla:
2     print(elemento)
```

```
Chile
8
4.15
Abril
3
5
2020
MOOC
```

ENCONTRAR UN ELEMENTO Dado que no podemos modificar una tupla, no existen operaciones como `append`, `pop`, `insert`, ó `remove`, pero sí podemos utilizar `index`.
El método `index` permite **retornar la posición** en que se encuentra un elemento dentro de la tupla. En caso que el elemento se encuentre repetido en la tupla, retorna la primera posición en

que lo encuentra. Este método supone que el elemento se encuentra efectivamente en la tupla. Si no es así, se produce un error, por lo que debemos estar seguros que el elemento existe antes de utilizar `index`.

1.4. Aplicaciones de tuplas

Una tupla es una colección de elementos **ordenados** y que podemos recorrer sistemáticamente, pero que por sobre todo es **inmutable**, lo que significa que una vez creada la tupla no podemos modificar sus contenidos.

Al momento de hablar de listas, utilizamos como ejemplo datos que eran típicamente homogéneos, pues queríamos hacer algo o extraer alguna información deducida a partir de ellos. Si bien en una tupla, también podemos almacenar elementos homogéneos, vamos a ver que en su uso práctico es bastante común utilizar elementos heterogéneos en las tuplas, esto es, elementos que no son necesariamente del mismo tipo.

Pensemos en estos ejemplos:

```
estudiantes = ['Aurora', 'Sebastián', 'Rafaela', 'Dario', 'Lisa',
 ↪  'Almendra', 'Camilo']
puntos = [67, 56, 48, 21, 60, 38, 89]
temperaturas = [16.8, 35.2, 27.9, 19.2, 19.8, 24.8, 34.3]
dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado',
 ↪  'Domingo']
```

tenemos nombres de estudiantes, puntajes, temperaturas, días de la semana, todos datos que posiblemente hemos leídos desde algún archivo o base de datos. Tiene mucho sentido utilizar listas para almacenar estos elementos, pues al momento de ir recorriendo el archivo, vamos agregando datos a las listas, usando el método `append`. Esto no lo podemos hacer con tuplas.

Entonces, ¿cómo podríamos usar tuplas? Vemos que en estos ejemplos, no hay realmente una asociación por ejemplo, entre el hecho que Rafaela está en la posición 2 de la lista `estudiantes`, y el valor que está en la posición 2 de la lista `puntos`, que es 48.

O, por ejemplo, si nos damos cuenta que la temperatura más alta ocurrió en el índice 1 de la lista `temperaturas`, para saber qué día ocurrió esa temperatura, tenemos que buscar el índice 1 de la lista `días`.

1.4.1. Ejemplos de uso de tuplas

BUSCAR PUNTAJE Supongamos que tenemos una lista de `str` con nombres de estudiantes, y una lista de sus correspondientes puntajes, que hemos leído de un archivo y se encuentra en la lista `puntos`. Si queremos buscar el puntaje de, digamos, Rafaela, tenemos que hacer una búsqueda en la lista de estudiantes, y luego acceder la posición correspondiente en la lista de puntos. Esto funciona de manera bastante eficiente, pero significa que tenemos mantener el registro de dos listas distintas.

```
busqueda = "Rafaela''
for i in range(len(estudiantes)):
    if estudiantes[i] == busqueda:
        pos = i
print(f"El puntaje de {busqueda} es {puntos[pos]}")
```

```
El puntaje de Rafaela es 48
```

En cambio, si tenemos nuestros datos de entrada escritos como tuplas dentro de una sola lista, podemos hacer una búsqueda más fácil de escribir, más clara de entender, y usando menos variables.

```
estud_puntos =[('Aurora', 67), ('Sebastián', 56), ('Rafaela', 48),
 ↪  ('Dario', 21), ('Lisa', 60), ('Almendra', 38), ('Camilo', 89)]
```

Podemos utilizar la búsqueda por elemento sobre la lista `estud_puntos`, donde cada elemento es una tupla. Para cada tupla, su posición 0 contiene un nombre, y su posición 1 contiene el puntaje correspondiente.

```
busqueda = "Rafaela''
for elemento in estud_puntos:
    if elemento[0] == busqueda:
        tupla = elemento
print(f"El puntaje de {tupla[0]} es {tupla[1]}")
```

```
El puntaje de Rafaela es 48
```

De esta manera, efectuamos la búsqueda del nombre sobre la posición 0 de cada elemento, que es una tupla, y cuando encontramos el elemento que buscamos, podemos acceder al puntaje correspondiente usando el índice 1 sobre el mismo elemento.
Esta solución funciona tan bien como la que usa únicamente listas, pero es más clara y evitar usar una variable adicional para recordar la posición. Es más, ahora cada tupla representa a un estudiante con su puntaje, en lugar de tener la información en dos listas separadas.

CONSTRUYENDO LISTAS DE TUPLAS: `ZIP` Probablemente una objeción a la solución anterior sería: bueno, ¿y cómo obtenemos los datos en formato de tuplas? Tendremos que hacer antes un ciclo que mezcle ambas listas, o preocuparse de construirlas así desde el inicio.
En Python, ya existe una función que mezcla estos datos de manera eficiente: la función `zip()`, que toma como entrada **dos listas**, y genera **una lista de tuplas** construidas a partir de cada elemento correspondiente de ambas listas. En caso que las listas no sean del mismo largo, `zip()` termina cuando una de la listas se acaba.

```
estudiantes = ['Aurora', 'Sebastián', 'Rafaela', 'Dario', 'Lisa',
 ↪  'Almendra', 'Camilo']
puntos = [67, 56, 48, 21, 60, 38, 89]
respuestas = ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']

for elemento in zip(estudiantes, puntos):
    if elemento[0] == busqueda:
        tupla = elemento
print(f"El puntaje de {tupla[0]} es {tupla[1]}")
```

```
El puntaje de Rafaela es 48
```

De esta manera, podemos utilizar una iteración sobre el resultado de la función `zip()` entre la lista `estudiantes`, y la lista `puntos`. Cada elemento generado por esta función es una tupla de

dos elementos, donde el primer elemento pertenece a la lista estudiantes, y el segundo pertenece a la lista puntos. Hemos resuelto el problema sin tener que construir una nueva lista con tuplas, sino que ésta se genera automáticamente a través de la función `zip`.

La función `zip` es bastante más poderosa aún. Podemos usarla con cualquier cantidad de estructuras sobre las que se pueda iterar, y lo que hace es generar una lista donde cada elemento es una tupla con un elemento correspondiente de cada estructura, hasta que la primera de ellas agota todos sus elementos. Esta construcción es muy útil cuando queremos unir datos correspondientes de distintas fuentes, y es muy eficiente porque no construye una lista nueva.

Podemos ver su efecto al imprimir cada uno de los elementos que resulta de construir un `zip` entre las listas `estudiantes`, `puntos` y `respuestas`.

```
estudiantes = ['Aurora', 'Sebastián', 'Rafaela', 'Dario', 'Lisa',
 ↪  'Almendra', 'Camilo']
puntos = [67, 56, 48, 21, 60, 38, 89]
respuestas = ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']

for elemento in zip(estudiantes, puntos, respuestas):
    print(elemento)
```

```
('Aurora', 67, 'a')
('Sebastián', 56, 'c')
('Rafaela', 48, 'c')
('Dario', 21, 'b')
('Lisa', 60, 'd')
('Almendra', 38, 'd')
('Camilo', 89, 'a')
```

Cada elemento generado es una tupla de 3 elementos, uno de cada lista. Notemos que la lista `respuesta` es más larga que estudiantes y puntos, pero el `zip` se termina cuando se acaba cualquiera de las listas.

Ahora, si queremos construir la lista completa, podemos tomar el resultado del `zip`, y utilizarlo para construir una nueva lista.

```
print(list(zip(estudiantes, puntos, respuestas)))
```

```
[('Aurora', 67, 'a'), ('Sebastián', 56, 'c'), ('Rafaela', 48, 'c'),
 ↪  ('Dario', 21, 'b'), ('Lisa', 60, 'd'), ('Almendra', 38, 'd'),
 ↪  ('Camilo', 89, 'a')]
```

Podemos usar `zip()` también para construir un ciclo que filtre elementos. Para nuestro ejemplo de temperaturas, podemos usar `zip()` para generar una secuencia de tuplas que asocie cada elemento de temperaturas con cada elemento de días, y en base a ellos, seleccionar solo aquellos tuplas cuyo primer elemento es mayor a 30, para agregar con `append()` a la lista de resultados. El resultado es la lista de tuplas con la información de los días calurosos.

```
temperaturas = [16.8, 35.2, 27.9, 19.2, 19.8, 24.8, 34.3]
dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado',
 ↪  'Domingo']
```

```
3
4  calurosos = []
5  for dia in zip(temperaturas,dias):
6      if dia[0] >= 30.0:
7          calurosos.append(dia)
8  print(f"Días calurosos: {calurosos}")
```

```
Días calurosos: [(35.2, 'Martes'), (34.3, 'Domingo')]
```

1.4.2. Packing y unpacking con tuplas

Es común utilizar tuplas para agrupar datos que están de alguna manera relacionados. Por ejemplo, podríamos usar tuplas para representar días, usando dos *strings* y dos enteros.

```
1  fecha = ("Sábado", 15, "Febrero", 2020)
2  print(fecha)
```

```
('Sábado', 15, 'Febrero', 2020)
```

A este proceso se le llama empaquetamiento, o *packing* de datos. Teníamos 4 variables distintas, y hemos juntado sus valores en un único objeto de tipo tupla.

Ahora, si queremos **extraer** los valores de esa tupla, podemos iterar sobre ella, pero si sabemos cuantos elementos hay podemos usar una tupla de variables para extraer o desempaquetar sus valores.

```
1  (dia_sem, dia, mes, año) = fecha
2  print(dia_sem)
3  print(dia)
4  print(mes)
5  print(año)
```

```
Sábado
15
Febrero
2020
```

Esta característica es muy usada en funciones cuando queremos retornar múltiples valores. Técnicamente una función siempre retorna un único valor, pero si queremos retornar más de uno podemos empaquetar los valores que queremos retornar en una tupla, y desempaquetarlos fuera de la función.

```
1  def seleccionar(nombre, todos):
2      for elemento in todos:
3          if elemento[0] == nombre:
4              return elemento        ## retornamos una tupla
5
6
7  estudiantes = ['Aurora', 'Sebastián', 'Rafaela', 'Dario', 'Lisa',
   ↪  'Almendra', 'Camilo']
```

```
puntos = [67, 56, 48, 21, 60, 38, 89]
respuestas = ['a', 'c', 'c', 'b', 'd', 'd', 'a', 'c', 'e', 'b']

res = seleccionar("Dario", zip(estudiantes, puntos, respuestas))
print(f"Resultado es {res}")

(e, p, r) = seleccionar("Dario",zip(estudiantes, puntos, respuestas))
print(f"El estudiante {e} tuvo {p} puntos y respondió {r}")
```

```
Resultado es ('Dario', 21, 'b')
El estudiante Dario tuvo 21 puntos y respondió b
```

Por ejemplo, tenemos una función que recibe un `zip()` con los estudiantes, puntos y respuestas, y selecciona una tupla de acuerdo al nombre. El resultado es una tupla, y podemos recibirlo directamente como tupla, o bien usar el desempaquetamiento para pasar los datos a 3 variables distintas.

1.4.3. Recomendaciones al usar tuplas

CREAR TUPLAS DE TAMAÑO UNO Un problema que suele haber al momento de crear una tupla de tamaño uno es que la notación `()` no da el resultado esperado. Si solo rodeamos nuestro único elemento con un `()`, se interpretará como una asignación de un valor unitario, y no de una tupla.

```
no_tupla_de_uno = ("Solitario")
print(type(no_tupla_de_uno))
print(no_tupla_de_uno)
```

```
<class 'str'>
Solitario
```

Podríamos pensar que usando el constructor de tuplas se puede resolver, pero si le pasamos, por ejemplo, un *string*, y ya que un `str` es en el fondo una secuencia de caracteres, lo que obtenemos es efectivamente una tupla, pero compuesta por muchos `str`, donde cada `str` es un carácter del *string* original.

```
tupla_string = tuple("Solitario")
print(type(tupla_string))
print(tupla_string)
```

```
<class 'tuple'>
('S', 'o', 'l', 'i', 't', 'a', 'r', 'i', 'o')
```

La manera de crear una tupla de tamaño uno parece poco intuitiva o convencional, pero consiste en poner una coma después del único elemento. De esta manera Python sabe que no es un valor único, y la única interpretación posible es que queremos crear una tupla.

```
tupla_de_uno = ("Solitario",)
print(type(tupla_de_uno))
print(tupla_de_uno)
```

```
<class 'tuple'>
('Solitario',)
```

TUPLAS CON ELEMENTOS MUTABLES Hemos repetido muchas veces que las tuplas son inmutables. Sin embargo, nadie nos ha prohibido **insertar elementos mutables** en una tupla. Si una tupla contiene una lista, esta lista es naturalmente un elemento mutable. No podemos cambiar la lista por otra, pero podemos aplicar operaciones sobre la lista, como `append`.

```
1 estudiante = ('Sebastián', [56, 27])
2 estudiante[1].append(90)
3 print(estudiante)
```

```
('Sebastián', [56, 27, 90])
```

Si tenemos el caso de una lista de tuplas, donde cada tupla contiene en su segundo elemento una lista, podemos aplicar a cada lista todas las operaciones que conocemos. Desde el punto de vista de cada tupla, no hemos modificado sus elementos. Las estructuras `list` siguen siendo las mismas. Solo hemos modificado la composición de cada lista, pues las listas sí son mutables.

```
1 estudiantes = [('Aurora', [67, 70, 29]), ('Sebastián', [56]), ('Rafaela',
   ↪ [48, 50, 10, 68]), ('Dario', [21, 30])]
2
3 estudiantes[0][1].sort()
4 estudiantes[1][1].append(90)
5 estudiantes[2][1].pop()
6 estudiantes[3][1].remove(21)
7 print(estudiantes)
```

```
[('Aurora', [29, 67, 70]), ('Sebastián', [56, 90]), ('Rafaela', [48, 50,
 ↪ 10]), ('Dario', [30])]
```

Esta es una pequeña "trampa" a la inmutabilidad de las tuplas, que tendrá consecuencias cuando queramos usarlas más adelante.

1.4.4. Listas vs tuplas

Entonces, ¿cuándo usamos listas o cuando usamos tuplas?

Ya hemos establecido que las listas son mutables: podemos agregar, eliminar, insertar elementos, y en las tuplas no es posible.

`list`	`tuple`
Mutable	Inmutable
Acceso eficiente a cada posición	Acceso más eficiente a cada posición
Cuando vamos a coleccionar o modificar datos del mismo tipo.	Agrupaciones de datos semánticamente relacionados.

Las listas son muy eficientes cuando queremos acceder a un elemento a través de su posición. No así cuando queremos buscar un elemento específico, pues tenemos que mirar cada uno de ellos hasta encontrarlo. Las tuplas no hacen esta búsqueda más eficiente, pero si permiten un acceso a la posición de cada elemento de manera más rápida aún que las listas. Esto tiene que ver con que al garantizar que los elementos de las tuplas no van a cambiar, las implementaciones de Python pueden optimizar la manera en que se guardan los datos para accederlos más rápidamente.

En general, la recomendación es usar listas cuando queremos trabajar con muchos elementos del mismo tipo, y potencialmente queremos modificarlos, agregar más elementos, o eliminar algunos. Por otro lado, las tuplas van a funcionar mejor cuando queremos agrupar elementos, que pueden ser de tipos distintos, pero que están semánticamente relacionados. Ya lo hicimos cuando teníamos el ejemplo de los estudiantes y formamos tuplas que reunían el nombre, el puntaje y una respuesta asociada al mismo estudiante; también las usamos cuando quisimos asociar temperaturas con días, y utilizamos tuplas que permitían juntar cada día con su temperatura correspondiente.

1.5. Colas y sus operaciones

Cuando hablamos de una cola, podemos recordar las filas que hacemos para pagar en un supermercado, para sacar dinero de un cajero automático, o lo que hacemos en el aeropuerto para poder registrarnos antes de viajar.

Precisamente ése el concepto de cola. Tal como las listas y las tuplas, son colecciones de datos, y por lo tanto son **estructuras de datos**.
La particularidad de las colas es que, además de tener todos sus elementos ordenados, las modificaciones ocurren **exclusivamente al inicio y al final** de ella. Típicamente los elementos que están al inicio de la cola son los primeros en ser eliminados de ella, por ejemplo, cuando somos atendidos en la caja del supermercado; y por otro lado los elementos nuevos suelen agregarse al final de la cola, como cuando llegamos hacer una fila y nos ponemos al final.

Si bien las colas comparten varias características con las listas y tuplas, las colas están diseñadas para ser más eficientes que una lista cuando se modifican elementos al inicio y al final, a cambio de funcionar de peor manera cuando se intenta acceder o modificar elementos entre medio de ellas.

1.5.1. La estructura `deque`

Si bien hay varias maneras de implementar colas en Python, en esta sección vamos a explorar una en particular, que es la clase `deque`. Esta clase no viene incorporada dentro del intérprete de Python, pero se puede importar a partir del módulo `collections`.

CREANDO UNA COLA VACÍA Podemos crear una nueva cola usando el inicializador `deque`.

```
1 from collections import deque
2 cola = deque()
```

Podemos imprimir el contenidos de la cola, en cuyo caso los muestra como si fuera una lista, pero rodeada del indicador `deque`.

```
1 print(cola)
```

```
deque([])
```

Al imprimir el tipo de dato, Python nos indica que el tipo es `deque` y que se encuentra definido dentro del módulo `collections`.

```
print(type(cola))
```

```
<class 'collections.deque'>
```

INICIALIZANDO UNA COLA CON ELEMENTOS Para crear colas podemos utilizar el inicializador de la clase `deque`. Podemos crear colas vacías, o crear colas con elementos, entregándole al inicializador alguna estructura **sobre la cual se pueda iterar**, esto es, que se pueda recorrer uno a uno sus elementos, como una lista o una tupla.

```
profesores = deque(['Jaime', 'Valeria', 'Cristian', 'Carla'])
print(profesores)
```

```
deque(['Jaime', 'Valeria', 'Cristian', 'Carla'])
```

De la misma manera que las otras estructuras secuenciales que hemos visto, podemos almacenar cualquier tipo de datos, y mezclar datos de distintos tipos dentro de las colas.

```
mezcla = deque(('Chile', 8, 4.15, 'Abril'))
sublista = deque([15, ['Arica', 9], 'Febrero'])
print(mezcla)
print(sublista)
```

```
deque(['Chile', 8, 4.15, 'Abril'])
deque([15, ['Arica', 9], 'Febrero'])
```

Dado que las colas no tienen una notación especial en la sintaxis de Python, al imprimir los contenidos de un `deque`, éstos se ven como una lista, pero rodeados del indicador de tipo `deque`.

CONSTRUIR UNA COLA A PARTIR DE OTRAS COLAS Podemos crear también colas mediante concatenación de otras colas utilizando también el operador de concatenación `+`.

```
otoño = deque(['Abril', 'Mayo', 'Junio'])
invierno = deque(['Julio', 'Agosto'])
frío = otoño + invierno
print(frio)
```

```
deque(['Abril', 'Mayo', 'Junio', 'Julio', 'Agosto’])
```

De la misma manera, podemos utilizado el operador $*$ para concatenar repetidas veces una cola consigo misma, como en estos ejemplos.

```
otoño = deque(['Abril', 'Mayo', 'Junio'])
invierno = deque(['Julio', 'Agosto'])
más_frío = invierno * 4
print(masfrio)
```

```
deque(['Julio', 'Agosto', 'Julio', 'Agosto', 'Julio', 'Agosto', 'Julio',
 ↪  'Agosto'])
```

1.5.2. Operaciones sobre `deque`

Una vez que tenemos la cola construir, podemos aplicar algunas operaciones sobre ella.

LONGITUD DE UNA COLA Si queremos conocer la cantidad de elementos en la cola, podemos utilizar la función `len()`.

```
mezcla = deque(['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC'])
largo = len(mezcla)
print(largo)
```

```
8
```

En este caso, tenemos una cola de 8 elementos.

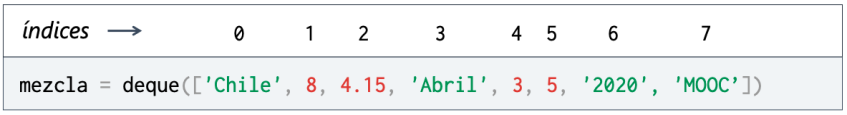

Figura 1.6: Una cola y sus índices. Notar que el primero es 0

BÚSQUEDA POR ÍNDICE Una cola también es una estructura secuencial. Esto significa que todos los elementos tienen una posición única, o índice, como se ve en la figura 1.6, y por lo tanto están ordenados entre sí. Usando el índice, podemos consultar por elementos en posiciones específicas de la cola, usando la notación con `[]`.

```
mezcla = deque(['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC'])
print(mezcla[0])
print(mezcla[4])
print(mezcla[7])
print(mezcla[-2])
```

```
Chile
3
MOOC
2020
```

Los índices empiezan **siempre** en la posición 0, por lo tanto si queremos obtener el primer elemento de la cola, preguntamos por la posición 0. Si queremos obtener el quinto elemento, preguntamos por el elemento en la posición 4, y así hasta que podemos preguntar por el elemento en la posición que equivale a la longitud de la cola menos uno. En este caso, la posición 7. En el caso particular de Python, podemos utilizar también índices negativos para consultar elementos desde la última posición, hacia la izquierda. Por ejemplo, si preguntamos por el índice -2, obtenemos el elemento en la penúltima posición de la cola.

Sin embargo, una consulta que **no es posible hacer en las colas**, es usar la notación de *slicing* para extraer porciones de la cola. Esto pareciera ser una desventaja respecto a otras estructuras como listas o tuplas, sin embargo, estas son decisiones de diseño al momento de construir la estructura, a favor de hacer más eficientes otro tipo de operaciones, ya que el concepto de una cola es el de una estructura cuyos principales operaciones se realizan al inicio y al final, y no entre medio de los extremos.
Si intentamos usar la notación de *slicing* en una cola, obtenemos un error.

```
mezcla = deque(['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC'])
print(mezcla[2:6])
print(mezcla[:3])
print(mezcla[5:])
print(mezcla[1:6:2])
```

```
TypeError: sequence index must be integer, not 'slice'
```

MODIFICAR ELEMENTOS DE UNA COLA Podemos modificar elementos individuales de una cola usando la notación de índices. Por ejemplo, podemos modificar los elementos en la posición 3 o en la posición 4, efectuando una asignación donde la notación `mezcla[indice]` se ubica al lado izquierdo de la asignación.

```
mezcla = deque(['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC'])
mezcla[3] = 'Mayo'
mezcla[4] += 5
print(mezcla)
```

```
deque(['Chile', 8, 4.15, 'Mayo', 8, 5, '2020', 'MOOC'])
```

RECORRIENDO COLAS Como otras estructuras secuenciales, en que podemos encontrar cada elemento usando su índice, podemos construir ciclos `for` sobre colas para recorrer de manera ordenada y sistemática la cola.
Podemos usar la instrucción `for i in range(len(cola))` para recorrer el largo de la cola. Esta instrucción nos permite utilizar una variable que avance desde la primera posición de la cola, la 0, hasta la última posición de la cola. En cada iteración, podemos usar la notación de índices para acceder a cada elemento de la cola. Esta manera de recorrer la cola es útil para cuando queremos encontrar un elemento en la cola y además nos interesa la posición en que se encuentra ese elemento.

```
mezcla = deque(['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC'])
for i in range(len(mezcla)):
    print(i, "-->", mezcla[i])
```

```
0 --> Chile
1 --> 8
2 --> 4.15
3 --> Abril
4 --> 3
5 --> 5
6 --> 2020
7 --> MOOC
```

De manera similar, podemos utilizar la notación `for elemento in cola`, más sencilla de escribir y que nos permite construir un ciclo en que la variable `elemento` toma, en cada iteración, el valor de un elemento de la cola, empezando siempre en el orden desde el primero hasta el último. Escribir la iteración de esta manera es muy conveniente cuando solamente queremos recorrer la cola, por ejemplo, para ver si cada elemento cumple una condición, o para buscar un elemento, pero no nos importa en qué posición se encuentra ese elemento.

```
1 mezcla = deque(['Chile', 8, 4.15, 'Abril', 3, 5, '2020', 'MOOC'])
2 for elemento in mezcla:
3     print(elemento)
```

```
Chile
8
4.15
Abril
3
5
2020
MOOC
```

AGREGAR Y SACAR DESDE EL FINAL El tipo de dato `deque`, en Python, está especialmente construido para efectuar operaciones de manera **muy eficiente en los extremos de la cola**. En particular, si queremos agregar y eliminar elementos en y desde el final de la cola, tenemos las operaciones respectivas `append` y `pop`, que funcionan de la misma manera que para las listas. [2] El método `append` recibe un argumento y lo agrega como elemento al final de la cola.

```
1 opciones = deque(['Sí', 'No'])
2 opciones.append('No sé')
3 print(opciones)
```

```
deque(['Sí', 'No', 'No sé'])
```

El método `pop` no recibe argumentos, sino que elimina el elemento que está en la última posición de la cola, y lo entrega como resultado.

[2] Podemos consultar todas las operaciones disponibles para colas en `https://docs.python.org/3/library/collections.html?highlight=deque#deque-objects`

```
cuando = deque(['Mañana', 'Pasado', 'Hoy'])
eliminado = cuando.pop()
print(cuando)
print(eliminado)
```

```
deque(['Mañana', 'Pasado'])
Hoy
```

AGREGAR Y SACAR DESDE EL INICIO Las operaciones que permiten agregar y eliminar elementos **desde el inicio de la cola**, son particulares a las colas. Éstas son, respectivamente, `appendleft` y `popleft`.
El método `appendleft` recibe un elemento y lo agrega como **primer elemento de la cola**.

```
opciones = deque(['Sí', 'No'])
opciones.appendleft('No sé')
print(opciones)
```

```
deque(['No sé', 'Sí', 'No'])
```

El método `popleft` no recibe argumentos, pero elimina el elemento que se encuentra en la primera posición de la cola, y lo entrega como resultado.

```
cuando = deque(['Mañana', 'Pasado', 'Hoy'])
eliminado = cuando.popleft()
print(cuando)
print(eliminado)
```

```
deque(['Pasado', 'Hoy'])
Mañana
```

Ambas operaciones son particulares a las colas, y son **más eficientes que las operaciones equivalentes en listas**.

INSERTAR EN CUALQUIER POSICIÓN En el caso de las colas, tenemos una operación para **insertar en cualquier posición**, pero no tenemos una operación para eliminar elementos a partir de una posición.

El método `insert` recibe una posición, y un elemento, e ingresa el elemento

```
opciones = deque(['Si', 'No', 'No sé'])
opciones.insert(2, 'Quizás')
print(opciones)
```

```
deque(['Si', 'No', 'Quizás', 'No sé'])
```

en la posición indicada de la cola.

ENCONTRAR Y EILMINAR UNA ELEMENTO Las colas también poseen operaciones que trabajan con elementos existentes en la ella.
El método `index` permite retornar la posición de un elemento que se encuentra en la cola. En caso que el elemento se encuentra más de una vez, retorna la primera posición en que lo encuentra. Este método supone que el elemento se encuentra en la cola. Si no es así, se produce un error.

```
opciones = deque(['Si','No','Quizás','No sé'])
print(opciones.index('Quizás'))
print(opciones)
```

```
2
```

El método `remove` recibe un elemento y lo elimina de la cola. Si el elemento se encuentra repetido, elimina solamente la primera ocurrencia. Al igual que `index`, este método supone que el elemento ya existe en la cola, de lo contrario se produce un error. La ejecución de `remove` **modifica la cola**, y **no entrega un valor de retorno**.

```
cuando = deque(['Mañana', 'Pasado', 'Hoy'])
cuando.remove('Pasado')
print(cuando)
```

```
deque(['Mañana', 'Hoy'])
```

1.6. Aplicaciones de colas

Si bien, en una cola podemos almacenar cualquier tipo de elementos, y mezclar elementos de distintos tipos, en la práctica es muy común que, tal como las listas, utilicemos colas para guardar elementos que sean todos del mismo tipo. Recordemos que las colas son estructuras secuenciales donde sus elementos están ordenados, pero lo más importante que es son estructuras **muy eficientes para modificar elementos en sus extremos**.

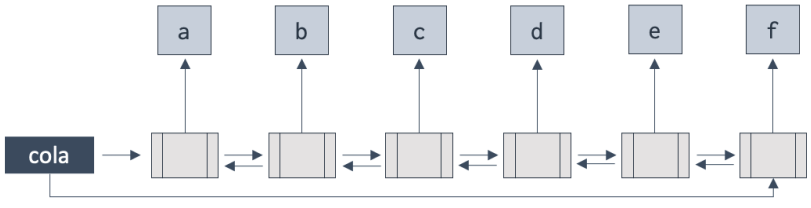

Figura 1.7: Estructura interna de una cola, como una lista doblemente ligada entre elementos

La figura 1.7 muestra cómo se ordenan los datos de una cola, luego de ejecutar:

```
cola = deque([a,b,c,d,e,f])
```

Una cola conoce la ubicación inmediata del **primer y último elemento**, y cada uno de sus elementos almacena un valor y contiene referencia a su **elemento sucesor** y a su **predecesor**.
No nos debería sorprender, entonces, que una aplicación típica de la estructura deque, sea precisamente el modelamiento y simulación de colas que ocurren en la vida real. Pensemos en

las cajas de atención de un supermercado. Cada una de ellas podría tener una cola con todos los clientes que desean ser atendidos. En ese contexto, algunas preguntas que quisiéramos hacer pueden ser

- ¿cuántos elementos (clientes) hay en la cola?
- ¿cuál es el siguiente elemento o cliente que debe salir de la cola para ser atendido?
- ¿qué hago cuando llega un nuevo elemento o cliente a la cola?

MODELANDO UNA FILA DE ATENCIÓN Supongamos que tenemos una cola con cinco personas, representadas por *string*s, que modelan una fila de atención.
El siguiente código permite construir la estructura `deque` que representa nuestra situación.

```
from collections import deque
cola = deque(['Sebastián', 'Rafaela', 'Darío', 'Lisa', 'Almendra'])
```

La figura 1.8 muestra cómo se ve esta cola.

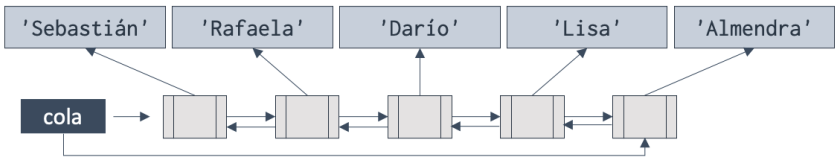

Figura 1.8: Ilustración de una cola con cinco persona representados por un `str`

Y con lo siguiente podemos consultar cuántas personas hay en la fila.

```
print(cola)
print(f"Hay {len(cola)} personas en la fila")
```

```
deque(['Sebastián', 'Rafaela', 'Darío', 'Lisa', 'Almendra'])
Hay 5 personas en la fila
```

Podemos modelar la llegada de una persona a la fila usando el método `append`.

```
cola.append("Camilo")
print(cola)
print(f"Hay {len(cola)} personas en la fila")
```

```
deque(['Sebastián', 'Rafaela', 'Darío', 'Lisa', 'Almendra', 'Camilo'])
Hay 6 personas en la fila
```

De esta manera, la persona que llega se agrega en la última posición de la fila, y nuestra fila ahora tiene 6 personas. La figura 1.9 muestra el estado de la cola luego de agregar a una persona al final.
Por otro lado, podemos modelar el hecho que una persona sea atendida. Esto significa que **la primera persona de la fila debería salir** de la cola, y esto lo modelamos usando el método `popleft`.

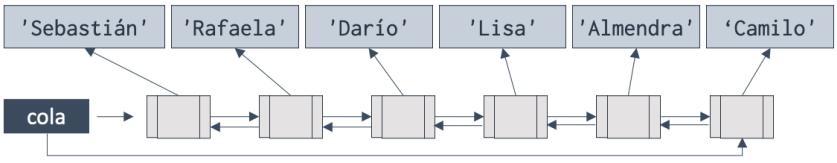

Figura 1.9: Ilustración de una cola luego de ejecutar un append

```
atendido = cola.popleft()
print(f"Cliente {atendido} ha pasado a la caja")
print(cola)
print(f"Hay {len(cola)} personas en la fila")
```

```
Cliente Sebastián ha pasado a la caja
deque(['Rafaela', 'Darío', 'Lisa', 'Almendra', 'Camilo'])
Hay 5 personas en la fila
```

Como consecuencia, ahora el primer elemento de la cola es el *string* `"Rafaela"`, y la fila tiene nuevamente 5 personas, como se ve en la figura 1.10

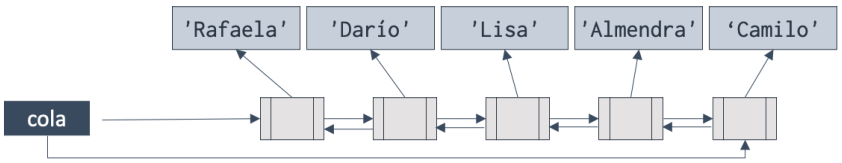

Figura 1.10: Ilustración de una cola luego de ejecutar un append

MODELANDO UN SUPERMERCADO CON MÚLTIPLES CAJAS Usando colas y otras estructuras, podemos modelar situaciones un poco más complejas. Supongamos que tenemos un supermercado con dos filas: una para una caja que atiene a clientes con más de 10 productos, y otra que es una caja "rápida" para atender clientes con menos de 10 productos.

Ya que ahora necesitamos saber cuántos productos tiene cada cliente, podemos modelar una cola de clientes que llega al sector de cajas usando un deque, donde cada cliente es una tupla que tiene un nombre de cliente, y la cantidad de productos que lleva. El supermercado, por otro lado, es una lista de dos deque, inicialmente vacíos.

```
from collections import deque
clientes = deque([('Sebastián', 30), ('Rafaela', 3), ('Darío', 8),\
                  ('Lisa', 5), ('Almendra', 27)])
supermercado = [deque(), deque()]
```

Lo que debemos hacer ahora es distribuir a los clientes en estas filas, de acuerdo a su orden de llegada. Para eso podemos sacar el primer cliente de la cola de clientes con popleft. Cada cliente es una tupla, donde el elemento 1 es la cantidad de productos. En base a ese valor podemos decidir si agregamos, usando append, al cliente al deque 0, o al deque 1 del supermercado.

```
while len(clientes) > 0:
    cliente = clientes.popleft()
    if cliente[1] > 10:
        supermercado[1].append(cliente)
    else:
        supermercado[0].append(cliente)
print(supermercado)
```

```
[deque([('Rafaela', 3), ('Darío', 8), ('Lisa', 5)]),
 deque([('Sebastián', 30), ('Almendra', 27)])]
```

Al imprimir el estado del supermercado, vemos que cada cola contiene los clientes que cumplen las condiciones.
Como desafío, podemos extender este problema usando otros criterios, por ejemplo, múltiples filas de cada tipo, y hacer que los clientes elijan la cola que tenga menos personas antes, o bien la cola cuya suma de productos sea menor. También podemos hacer más complejos a los clientes agregando elementos a la tupla que indiquen, por ejemplo, si es cliente preferente del supermercado, o si posee alguna condición especial como tercera edad o embarazo que podría hacerlos pasar a una cola dedicada.

1.7. Escogiendo estructuras de datos

Hemos conocido tres estructuras de datos secuenciales: listas, representadas por el tipo `list` de Python; tuplas, representadas por el tipo `tuple`; y colas, representadas por el tipo `deque`. La figura 1.11 resumen estas estructuras.

Todas ellas, por el hecho de ser secuenciales, comparten el hecho que mantienen un orden entre los elementos y, a través de un valor de posición que llamamos índice, podemos acceder a cada elemento. Además de ello, las estructuras poseen operaciones para consultar y manipular los elementos que almacenan.

`list`	`tuple`	`deque`
`lista = [67,56,48,21]`	`tupla = (67,56,48,21)`	`cola = deque([67,56,48,21])`
Mutalee ✓	Mutable ✗	Mutable ✓
Indexable ✓	Indexable ✓	Indexable ✓
Slicing ✓	*Slicing* ✓	*Slicing* ✗
Agregar/eliminar al inicio ✓	Agregar/eliminar al inicio ✗	Agregar/eliminar al inicio ✓✓✓
Agregar/eliminar al final ✓✓✓	Agregar/eliminar al final ✗	Agregar/eliminar al final ✓✓✓
Insertar/eliminar en posición `i` ✓	Insertar/eliminar en posición `i` ✗	Insertar/eliminar en posición `i` ✓
Leer cualquier posición ✓✓✓	Leer cualquier posición ✓✓✓	Leer cualquier posición ✓
Útil para elementos homogéneos	Útil para elementos heterogéneos	Útil para elementos homogéneos con modificaciones en los extremos

Cuadro 1.1: Comparación de característica de `list`, `tuple`, y `deque`

La tabla 1.1 resume las características de cada estructura. Recordemos que de las tres, las tuplas son las únicas cuyos elementos no pueden ser modificados, aunque bien pueden contener elementos que sí son mutables (como una lista).

Al ser estructuras secuenciales, las tres permiten acceder a elementos a través de su índice. Solo listas y tuplas permiten extraer eficientemente porciones usando la notación de *slicing*.

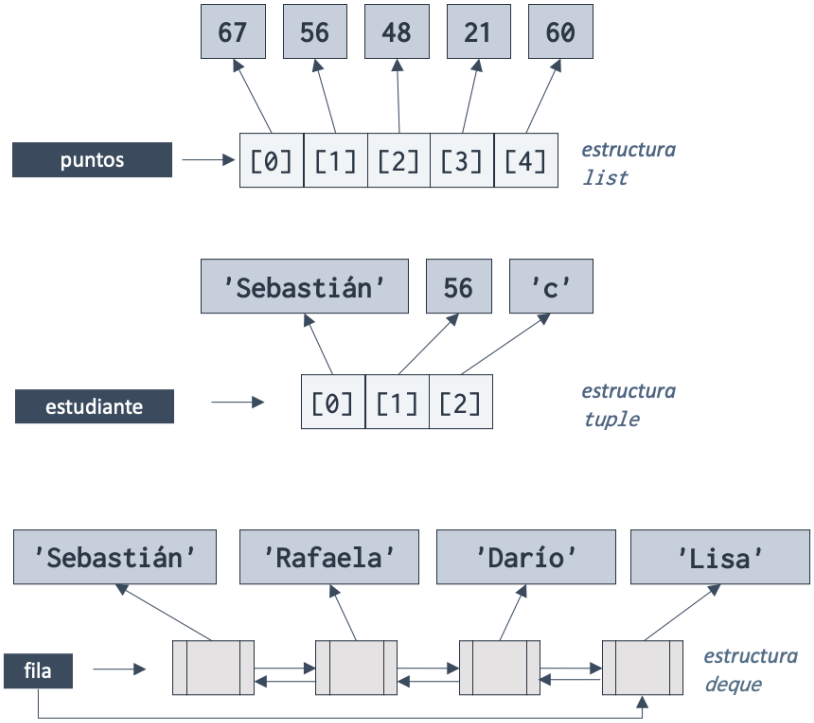

Figura 1.11: Estructuras secuenciales: `list`, `tuple`, `deque`

Pero lo más interesante es la eficiencia de las operaciones. Cuando hablamos de eficiencia nos referimos a cuán rápido se ejecutan estas operaciones cuando tenemos muchos elementos en la estructura. En general, diremos que una operación es muy eficiente cuando la cantidad de elementos en la estructura no influye, o influye muy poco en el tiempo de ejecución de la operación. Diremos que la operación no es eficiente cuando va tomando más tiempo a medida que aumenta la cantidad de elementos en la estructura.

Esto tiene que ver con el objetivo para el cuál están hechas estas estructuras. En general, no existe una estructura de datos que sea la mejor para todos los casos. La elección siempre depende de qué tipo de operaciones necesitamos hacer con los datos, o qué tipo de operación será la más común.

Pues bien, vamos a ver que las listas son muy eficientes para **agregar o eliminar elementos al final** de ellas, en la última posición. También son muy eficientes para **acceder a cualquier posición usando el índice** de un elemento. Sin embargo, no son tan eficientes para agregar elementos al inicio o en alguna parte intermedia de ellas, que no sea la última posición.

Las tuplas, por otro lado, son muy eficientes, al igual que las listas, para acceder a cualquier posición. Las operaciones de modificación no son aplicables, pues se trata de una **estructura inmutable**. Sin embargo debemos recordar que la utilidad de las tuplas es la de almacenar elementos que están relacionados entre sí y que no necesitan ser modificados.

Finalmente, las colas fueron diseñadas para ser muy eficientes al momento de **agregar o eliminar elementos de sus extremos**, tanto del inicio, como del final. Sin embargo, no funcionan tan bien cuando queremos acceder un elemento en una posición cualquiera.

¿Por qué ocurren estas diferencias de eficiencia? Comparemos un poco la manera en que están construidas las listas y las colas. Dejaremos de lado las tuplas, pues su construcción es muy similar a las listas. El siguiente código crea una lista y una cola con los mismos elementos.

```
lista = [67,56,48,21,60]
cola = deque([67,56,48,21,60])
```

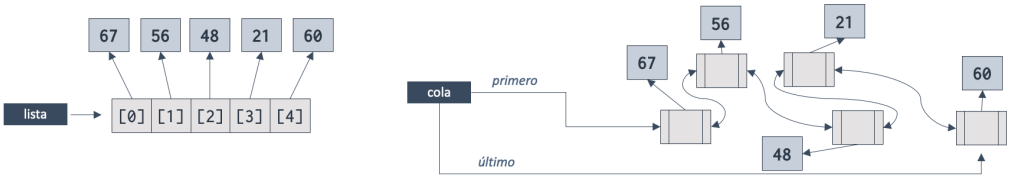

Figura 1.12: Estructuras secuenciales: `list`, `tuple`, `deque`

La figura 1.12 muestra la composición interna de cada estructura. La característica principal de las listas es que todos sus elementos están uno al lado del otro, a partir del 0. Esto significa que si sabemos dónde empieza la lista, es muy rápido llegar a cualquier posición k, pues tengo que sumar k a la posición inicial. De la misma manera, agregar un elemento al final es moverse hasta la última posición y pegar un elemento nuevo al lado del último.

Por otro lado, las colas están compuestas por múltiples unidades conectadas entre sí, no necesariamente una al lado de la otra. Esto significa que para llegar a la posición k, la cola debe empezar por la primera posición, y **buscar k veces** la posición de la unidad siguiente hasta encontrar la posición deseada. En esto la manera en que están construidas las listas otorga una ventaja.

Pero, ¿qué pasa si queremos agregar un elemento al inicio o entre medio de la lista? Encontrar la ubicación es sumamente rápido, pero como las listas exigen que cada ubicación esté contigua una a la otra, cuando queremos insertar un elemento en la posición k, hay que **copiar todos los elementos restantes una posición a la derecha**, de manera de hacer espacio para el elemento nuevo. Igual que cuando estamos sentados en el cine, y alguien quiere sentarse en un asiento entre medio de dos personas. Entonces, cuando queremos insertar un elemento en la primera posición, la lista desplaza todos sus elementos una posición a la derecha. Esto es más lento mientras más elementos hay en la lista.

En el caso de las colas, insertar un elemento tanto al inicio como al final, es muy fácil. La cola conoce la ubicación del primer elemento. Insertar al inicio significa crear un elemento nuevo, indicar que su siguiente elemento es el segundo, y ahora modificar la cola para que el primer elemento sea el que acabamos de agregar. Lo mismo ocurre si queremos agregar un elemento al final. No es necesario modificar ningún elemento más.

Insertar en cualquier posición no es particularmente eficiente en ninguna estructura, pero por razones distintas. Para las listas es fácil encontrar la posición donde queremos insertar, pero la inserción misma es lenta. En el caso de las colas, la inserción es muy sencilla, pero encontrar la posición es lento.

EFICIENCIA DE BÚSQUEDA EN LISTAS VERSUS COLAS Vamos a realizar un experimento. El siguiente código crea un `deque` y una lista, ambos de 10 millones de números. No importa qué números sean. Vamos a medir cuánto demora cada estructura en **encontrar el elemento en la posición 5 millones**, o sea, justo al medio.

```
from collections import deque
from time import time
number_deque = deque(range(10000000))
```

```
number_list = list(range(10000000))

start_time = time()
number_deque[5000000]
finish_time = time()
deque_time = finish_time - start_time
print(f"Buscar el elemento 5000000 en el deque demoró {deque_time:.6f}
 ↪  seg")

start_time = time()
number_list[5000000]
finish_time = time()
list_time = finish_time - start_time
print(f"""Buscar el elemento 5000000 en la lista demoró {list_time:.6f}
 ↪  seg")
print(f"La búsqueda en deque fue {deque_time/list_time:.2f} veces el
 ↪  tiempo de list.")
```

```
Buscar el elemento 5000000 en el deque demoró 0.004431 seg
Buscar el elemento 5000000 en la lista demoró 0.000060 seg
La búsqueda en deque fue 74.35 veces el tiempo de list
```

Los tiempos precisos van a variar en cada computador que se ejecuten, y en cada ejecución. Sin embargo, los resultados serán equivalentes. En este caso podemos observar que para **encontrar el elemento que se encuentra en la mitad** de la estructura, el deque se demoró casi 75 veces más que la lista para llegar a la misma posición.

EFICIENCIA DE ELIMINACIÓN EN LISTAS VERSUS COLAS Este experimento consiste en sacar los primeros 1000 elementos del inicio de un deque con `popleft()`, versus sacar los primeros 1000 del inicio de una lista con `pop(0)`.

```
number_deque = deque(range(10000000))
number_list = list(range(10000000))

start_time = time()
for i in range(1000):
    number_deque.popleft()
finish_time = time()
deque_time = finish_time - start_time
print(f"Sacar los primeros 1000 de deque demoró {deque_time:.6f} seg")

start_time = time()
for i in range(1000):
    number_list.pop(0)
finish_time = time()
list_time = finish_time - start_time
print(f" "Sacar los primeros 1000 de lista demoró {list_time:.6f} seg")
print(f"La extracción en list fue {deque_time/list_time:.2f} veces el
 ↪  tiempo de deque.")
```

```
Sacar los primeros 1000 de deque se demoró 0.000152 seg
Sacar los primeros 1000 de lista se demoró 7.044542 seg
La extracción en list fue 46457.47 veces el tiempo de deque.
```

En esta caso el deque muestra toda su eficiencia, ya que la lista demora 7 segundos en hacerlo, y el deque demora menos de un milisegundo. El deque fue **46 mil veces más rápido**.

Por lo tanto, dependiendo de la operaciones que vamos a utilizar con nuestra estructura, podemos ver que la elección de la estructura a utilizar puede ser muy importante en el rendimiento de nuestros programas.