Listas en Python

Python tiene cuatro tipos de datos incorporados que sirven para guardar colecciones de datos: lista, tuple, set y diccionario. Estos tipos difieren en cuanto a función, desempeño y maleabilidad.

Por ahora, nos enfocaremos en las listas Python, cómo funcionan y cómo pueden ser manipuladas para resolver tus necesidades. Pero primero, revisemos lo básico detrás de lo que es una lista y de qué es capaz.

Una lista en Python es:

1. **Ordenada**: esto quiere decir que los elementos dentro de ella están indexados y se accede a ellos a través de una locación indexada.
2. **Editable**: los elementos dentro de una lista pueden editarse, añadir nuevos o eliminar los que ya tiene.
3. **Dinámica**: las listas pueden contener diferentes tipos de datos y hasta de objetos. Esto significa que pueden soportar paquetes multidimensionales de datos, como un array o muchos objetos.
4. **No única**: esencialmente, esto quiere decir que la lista puede contener elementos duplicados sin que arroje un error.

Si te has familiarizado con la programación, podría parecer que la lista en Python es similar a un array en otros lenguajes. Seamos claros al respecto: son muy distintos entre ellos. Python tiene un tipo de datos array nativo por esa misma razón.

¿Cómo crear una lista en Python?

Crear una lista en Python es muy simple, y gracias que cuenta con dos alternativas para hacerlo, tenemos mucho más control sobre cómo y dónde es posible generar y gestionar una lista. El primer método para crear una lista en Python es utilizando la notación de corchetes **[ ]**. Revisemos cómo lograrlo.

```
serpList = ["Boa", "Pitón", "Culebra venenosa", "Víbora Rayada"]
```

Eso es todo, ya que la notación de corchetes es bastante directa: simplemente envuelves el contenido que deseas añadir a tu lista con los corchetes cuadrados. La segunda manera es utilizando el constructor de lista; en este caso, la sintaxis es un poco diferente. Mira este ejemplo.

```
serpList = list(("Boa", "Pitón", "Culebra venenosa", "Víbora Rayada"))
```

En este caso, el resultado de ambas líneas de código equivaldría más o menos al mismo, y sería idéntico en cada aspecto.

```
print(serpList)
```

Imprimiría este resultado:

```
["Boa", "Pitón", "Culebra venenosa", "Víbora Rayada"]
```

Sin embargo, existen salvedades a considerar cuando se trabaja con colecciones de datos más dinámicas. Por ejemplo, convertir una colección de datos diccionario usando el constructor de lista desglosaría la información a solo las claves de esa colección. En contraste, la notación de corchetes mantendría todas las claves y valores como un elemento en la lista.

Revisemos la lista serpList anterior y convirtámosla en un objeto diccionario.

```
serpList = {"Boa": "Serpiente Constrictora", "Pitón": "Serpiente Constrictora", "Cobra": "Culebra venenosa", "Víbora Rayada":'
```

Si el diccionario de arriba se convirtiera en una lista utilizando el constructor, el resultado filtraría todos los valores en este diccionario.

```
list(serpList)
```

En su lugar, crearía una lista con esas serpientes (claves) y eliminaría sus respectivas designaciones de seguridad (valores). Al imprimir el resultado de esta acción arrojaría lo siguiente.

```
print(serpList)["Boa", "Pitón", "Cobra", "Víbora Rayada"]
```

Este resultado contrasta con la notación de corchetes, que toma este objeto diccionario y lo preserva como un solo elemento en la lista. Veamos cómo luce.

```
[serpList]print(serpList)[{"Boa": "Serpiente Constrictora", "Pitón": "Serpiente Constrictora", "Cobra": "Culebra venenosa", "\
```

Ahora podemos ahondar en este concepto un poco más. Enseguida revisaremos los tipos de datos que son compatibles con las listas Python.

Tipos de datos para listas en Python

Las listas en Python son mucho más amigables que otras al mezclar tipos de datos. Una lista Python te permite tener otras listas que contienen varios elementos del mismo tipo o una combinación de diferentes tipos de datos. Es más: las listas son compatibles con cualquier tipo de datos, incluyendo enteros, cadenas, booleanos, listas anidadas y objetos.

Esto quiere decir que tienen el potencial de ser una herramienta muy poderosa en tu banco de conocimiento Python. Ahondemos más en ella echando un vistazo a cómo manipular tus listas en Python.

¿Cómo manipular las listas en Python?

Python tiene algunas funciones nativas que podemos usar para ayudarte a reafirmar un mejor control sobre cómo usarlas. Revisemos algunas de las funciones más comunes que se relacionan con las listas.

Funciones nativas de Python

Las funciones más relevantes son la función de longitud o length **len()** y las funciones de tipo **(type)**, que hacen lo que su nombre indica. Por ejemplo, en el caso de length, calcula el número de elementos dentro de una lista.

Es importante que consideres que si usas esto en una lista con diccionarios, listas anidadas o similares, la función solo arrojará cada elemento de hasta arriba. Eso significa que la función length no revisará ni contará los elementos dentro de listas anidadas, únicamente la lista anidada.

Veamos cómo trabaja la función len() con listas, utilizando nuestra serpList de arriba.

```
serpList = ["Boa", "Pitón", "Víbora Rayada"]
print(len(serpList))
```

El código de arriba arroja el resultado 3, mientras que el código de abajo también arrojará el resultado de 3.

```
serpList = [{"Boa", "Pitón", "Víbora Rayada"},{"Serpiente Constrictora", "Serpiente Constrictora", "Culebra Inofensiva"},"Serp
```

La siguiente función en la agenda es la de tipo **type**, que toma un objeto y determina su tipo de datos. No hay mucho detrás de cómo funciona en lo que a listas concierne, pero es muy útil para confirmar que una lista es realmente del tipo **list**. Es más: el tipo de elementos dentro de la lista no afectan su tipo. Veámosla en acción.

Ambas declaraciones de lista evalúan al mismo resultado.

```
serpList = ["Boa", "Pitón", "Víbora Rayada"]
serpList = [{"Boa", "Pitón", "Víbora Rayada"},{"Serpiente Constrictora", "Serpiente Constrictora", "Culebra Inofensiva"},"Serp
```

Si imprimes el tipo o cualquiera de estas declaraciones, tendrás el mismo resultado.

```
print(type(serpList))<class 'list'>
```

Operadores de Python

Los operadores nativos de Python abren una dinámica totalmente nueva para interactuar con listas Python. Puedes usarlos para añadir, eliminar, concatenar, fusionar, imprimir y más. Primero, revisemos algunos de los operadores más populares que se utilizan en estas listas.

El operador del símbolo de más **+** (concatenar) sirve para fusionar una o más listas, creando una más larga gracias a la combinación de las listas deseadas.

```
serpList = ["Boa", "Pitón", "Víbora Rayada"]
print(serpList + ["182 centímetros", "121 centímetros", "30 centímetros"])
```

El código de arriba resultaría en lo siguiente:

```
serpList = ["Boa", "Pitón", "Víbora Rayada", "182 centímetros", "121 centímetros", "30 centímetros"]
```

A continuación, hablaremos del operador **slice** (rango), que declara un rango de elementos de tu lista, según su locación indexada. De esa manera, accederás a los elementos de la misma manera que accedes a un elemento en tu lista, usando la notación de corchetes. Así se ve en acción.

```
print(serpList[0:4])
```

La línea de código de arriba arrojará el resultado de los primeros tres elementos en la lista, comenzando en el índice de 0 y terminando antes del de 4. En otras palabras, el punto de inicio del operador slice es inclusivo, y el punto final es exclusivo.

```
["Boa", "Pitón", "Víbora Rayada"]
```

El operador de multiplicación ***** (repetir) sirve para dictar el número de veces que una acción en particular debe repetirse. En el caso de una lista, esto permite una multitud de funcionalidades que tienen que ver con la manipulación de los elementos de una lista. Imprimamos un ejemplo.

```
print(serpList * 3)
```

Esta línea de código imprimiría el resultado de la ecuación provista, una lista que contiene la misma lista de elementos repetidos de forma secuencial, tres veces.

```
["Boa", "Pitón", "Víbora Rayada", "Boa","Pitón", "Víbora Rayada", "Boa", "Pitón", "Víbora Rayada"]
```

El operador de asignar **=** (cambio) cambiará directamente el valor de un elemento de la lista, de la lista de elementos o hasta de la lista completa. Revisemos un ejemplo de esto, con la lista serpList.

```
serpList = ["Boa", "Pitón", "Víbora Rayada"]serpList = serpList * 3
```

La línea de código de arriba toma el valor original de serpList, lo multiplica por tres, y le asigna el valor de la ecuación a sí misma. El resultado es que la lista ahora combina nueve elementos, los primeros tres repetidos tres veces.

```
print(serpList)
```

Que arrojaría lo siguiente.

```
["Boa", "Pitón", "Víbora Rayada", "Boa", "Pitón", "Víbora Rayada", "Boa", "Pitón", "Víbora Rayada"]
```

Métodos para listas en Python

Como sucede con la mayoría de los lenguajes de programación, un método es típicamente llamado como el objeto contra el que deseas correrlo. Muchos métodos, pero no todos, pueden tomar un argumento cuando se invocan. Llamar a un método es simple y generalmente se hace al encadenarlo al objeto con un **.** que funciona como el vínculo entre ambos.

object.method(argument)

Ya que cubrimos el tema, revisemos la sintaxis de la llamada a un método que se refiere a una lista Python. Primero, veamos tres de los métodos más populares para interactuar y modificar una lista en Python.

Si únicamente quieres añadir un nuevo elemento al final de tu lista, puedes usar el método agregar .**apend** para sumarlo.

```
serpList = ["Boa", "Pitón", "Víbora Rayada"]
serpList.append("Cobra")
```

Esa línea de código agrega una nueva cadena a la lista. El resultado, si lo imprimes en la pantalla, luciría así.

```
print(serpList)["Boa", "Pitón", "Víbora Rayada", "Cobra"]
```

El siguiente método más popular es el método de índice **index** que sirve para arrojar el índice de un valor específico. Por ejemplo, veamos qué sucede si apuntamos al elemento cadena «Cobra».

```
serpList.index("Cobra")
```

El resultado de esta línea de código sería el valor de índice de la lista de elementos. En este caso, sería índice 3.

Finalmente, vamos al método eliminar **remove** que se utiliza para apuntar un elemento específico de la lista y eliminarlo. Este método permite mucho control, siempre y cuando sepas exactamente a lo que te diriges y no tienes dudas de que está en la lista. Echemos un vistazo a cómo usarlo antes de cerrar este artículo.

```
serpList.remove("Cobra")
```

Esa línea de arriba elimina con éxito el elemento que coincide con el objetivo descrito. En este caso, el elemento de la lista corresponde a la cadena «Cobra». Por lo tanto, al imprimir la lista después de invocar este método resultaría en lo siguiente.

```
print(serpList)["Boa", "Python", "Víbora Rayada"]
```

¿Cómo ordenar las listas en Python?

Casi todos los conceptos dentro de los lenguajes de programación son abstracciones de actividades o funciones que usamos en nuestra realidad cotidiana; las listas son uno de esos casos. Todos los días, a veces ya de manera inconsciente, debemos ordenar un conjunto de objetos, por ejemplo del más pequeño al más grande, por orden alfabético, por color, entre otros. En este caso contamos con dos opciones para ordenar listas en Python.

Método incorporado list.sort()

Es un método que se encuentra incorporado dentro del abanico de métodos inherentes del tipo de objeto lista de Python. Para utilizarlo basta con invocarlo dentro de una instancia de lista. El valor que retorna es **None** y la lista ordenada es almacenada en la variable original. A continuación te mostramos un ejemplo dentro del prompt de Python.

```
>>>> a = [13,5,7,2,1]

>>> a.sort()

>>> a

[1, 2, 5, 7, 13]
```

Función incorporada sorted()

Esta función retorna una nueva lista ordenada, es decir, no modifica la variable original. Esta función es aplicable a cualquier iterable y no solamente a objetos de tipo lista.

```
>>>> sorted([13,5,7,2,1])

[1, 2, 5, 7, 13]

>>> sorted({1: 'D', 2: 'B', 4: 'E', 5: 'A', 0: 'e'})

[0, 1, 2, 4, 5]
```

Orden descendente

Tanto en **sorted()** como en **list.sort()** tenemos disponible un parámetro **reverse** con un valor booleano que nos permite indicar cuando queremos realizar un ordenamiento de forma descendente.

```
>>>> sorted([13,5,7,2,1], reverse=True)

[13, 7, 5, 2, 1]

>>> sorted({1: 'D', 2: 'B', 4: 'E', 5: 'A', 0: 'e'}, reverse=True)

[5, 4, 2, 1, 0]
```

Funciones Key

Key es un parámetro que tenemos disponible en **sorted()** y en **list.sort()** y sirve para especificar una función que se invocará en cada elemento de la lista, justo antes de realizar comparaciones.

En el siguiente ejemplo se muestra la diferencia entre ordenar cadenas distinguiendo entre mayúsculas y minúsculas, y ordenar cadenas sin distinguir mayúsculas y minúsculas.

```
>>>> sorted("Ana añora estar con Emma y Carla".split())

['Ana', 'Carla', 'Emma', 'añora', 'con', 'estar', 'y']

>>> sorted("Ana añora estar con Emma y Carla".split(), key=str.lower)

['Ana', 'añora', 'Carla', 'con', 'Emma', 'estar', 'y']
```

El parámetro **key** debe ser una función de un solo argumento y que retorne una clave que pueda usarse con fines de ordenamiento. Esto ayuda a que el algoritmo sea rápido ya que dicha función se llama exactamente una vez por cada registro en el iterable a ordenar.

A continuación te mostramos un ejemplo utilizando la palabra reservada **lambda** de Python para crear una función lambda que es usada como parámetro **key**.

```
>>>> student_tuples = [... ('Alice', 'B', 15),... ('Bob', 'C', 10),... ('Carl', 'A', 12),... ]

>>> sorted(student_tuples, key=lambda student: student[0])[('Alice', 'B', 15), ('Bob', 'C', 10), ('Carl', 'A', 12)]

>>> sorted(student_tuples, key=lambda student: student[1])[('Carl', 'A', 12), ('Alice', 'B', 15), ('Bob', 'C', 10)]

>>> sorted(student_tuples, key=lambda student: student[2])[('Bob', 'C', 10), ('Carl', 'A', 12), ('Alice', 'B', 15)]
```

Bibliografía:

- Extraído de <<Listas en Python: qué son, cómo crearlas y ordenarlas>>. 28 de septiembre de 2022. Consultado el 28 de noviembre del 2022