Introducción a Estructuras de Datos

¡Comencemos!

¿Qué aprenderás?

- Reconocer los tipos de datos usados en el entorno Python y su uso para la construcción de un programa, las diferencias entre ellos y sus usos más típicos.

Introducción

En la primera parte de la unidad, aprendimos los fundamentos y parte de la sintaxis más básica del lenguaje Python. A lo largo de esta lectura aprenderemos acerca de las estructuras de datos. donde estas, al igual que las variables, también son contenedores, pero la principal diferencia es que estas tienen más dimensiones y pueden contener más de un valor a la vez.

Dentro de las estructuras nativas de Python están las Tuplas, los Sets, las Listas y los Diccionarios. Y en este capítulo nos enfocaremos principalmente en las Listas que son una de las estructuras de datos más importantes en Python.

Introducción a Listas

Las listas son contenedores que permiten almacenar múltiples datos. En lugar de guardar un solo número o un solo string en una variable, ahora podemos almacenar en un **objeto** varios valores simultáneamente, tal como se observa a continuación:

```
# Estos elementos pueden ser de un mismo tipo de datos, por ejemplo solo strings:
animales = ['gato','perro','raton']

# O pueden ser de distintos tipos de dato.
lista_heterogenea = [1, "gato", 3.0, False]
```

Los elementos de la lista se pueden modificar, ya sea cambiando los valores de éstos, agregando o eliminando elementos. Por eso es que se dice que las listas son **mutables**.

Las listas son muy utilizadas dentro de la programación, y son útiles para resolver diversos tipos de problemas; como por ejemplo, si se quiere hacer una aplicación web, se puede traer los datos de la base de datos a una lista y operar los datos desde ahí. También son útiles para leer archivos y crear gráficos entre múltiples otras funciones.

En esta sesión, se enseñarán las propiedades fundamentales de las listas, por lo que entender esto es fundamental, ya que al combinar estas propiedades con aspectos más avanzados del lenguaje Python, es lo que nos permitirá crear aplicaciones cada vez más potentes.

Definir y mostrar una lista

Para crear una lista utilizaremos la siguiente sintaxis:

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

Se definen con los paréntesis de corchete `[]`; Todo lo que esté dentro de los `[]`, separados por coma, son los elementos de la lista.

Lo más sencillo que podemos hacer con una lista es mostrar sus elementos. Para esto podemos ocupar `print` o llamar a su objeto contenedor.

```
print(a) # mostramos los valores de la lista a definida arriba
print([1, 2, 'hola', 4]) # mostramos directo una lista
```

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

```
[1, 2, 'hola', 4]
```

Índices

Cada elemento en la lista tiene una posición específica, la que se conoce como índice, el que permite acceder al elemento que está dentro de la lista.

En Python **los índices parten en cero** y van hasta `n - 1`, donde `n` es la cantidad de elementos en la lista.

Por ejemplo: En una lista que contiene 4 elementos, el primer elemento está en la posición cero, y el último en la 3.

Para acceder al elemento de una posición específica de la lista, simplemente se debe escribir el nombre de la lista, y entre paréntesis de corchetes, el índice de la posición.

```
colores = ["verde", "rojo", "rosa", "azul"]
print(colores[0])
print(colores[1])
print(colores[3])
```

```
verde
rojo
azul
```

Índices más allá de los límites

En caso de que el índice sea mayor o igual a la cantidad de elementos en la lista, Python arrojará un `IndexError: list index out of range`, que indica que el índice se posiciona fuera del rango de la lista.

```
colores[8]
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call
last)
<ipython-input-4-cf0cdf800f9c> in <module>()
----> 1 colores[8]
IndexError: list index out of range
```

Índices negativos

Los índices también se pueden utilizar con números negativos y de esta forma referirse a los elementos desde el último al primero.

En Python, el índice del último elemento también se puede denotar con -1 y el primer elemento corresponde al elemento `-n`

```
a = [1, 2, 3, 4, 5]
a[-1]
```

```
5
```

ARGV

Hasta el momento vimos que `input()` es una instrucción que nos permite ingresar datos por parte del usuario para poder ser utilizados dentro de nuestros programas. Ahora presentaremos `argv,` los cuales son argumentos que se pueden ingresar desde la terminal al momento de ejecutar nuestro programa.

Ejecución guiada 1

Mini Presentador

Aplicar como combinar la interpolación con `argv` para poder importar datos a Python. Sigamos los siguientes pasos:

1. Crear el archivo `argumentos.py`.

2. Lo primero que haremos en nuestro archivo, es importar el módulo que nos permite rescatar argumentos desde el terminal:

```
import sys
```

3. Dentro de este módulo, llamaremos a la propiedad `argv`, la cual corresponde a la lista de los argumentos introducidos en el terminal.

   El primer argumento (cuando se ejecuta un script) **es el nombre del mismo**, y se encuentra ubicado en la posición **0** de la lista de argumentos.

   Todo lo que se escriba después del nombre del script, corresponderá a los argumentos siguientes.

   Crearemos una variable **nombre**, a la cual le asignaremos el argumento en la posición `1` de la lista de argumentos y la variable apellido con la posición `2`, de la siguiente forma:

```
nombre = sys.argv[1]
apellido = sys.argv[2]
```

4. Ahora visualicemos los resultados ingresados agregando lo siguiente:

```
print(f"Mi nombre es {nombre}")
print(f"Mi apellido es {apellido}")
print(f" nombre de este archivo es {sys.argv[0]}")
```

El código completo se ve así:

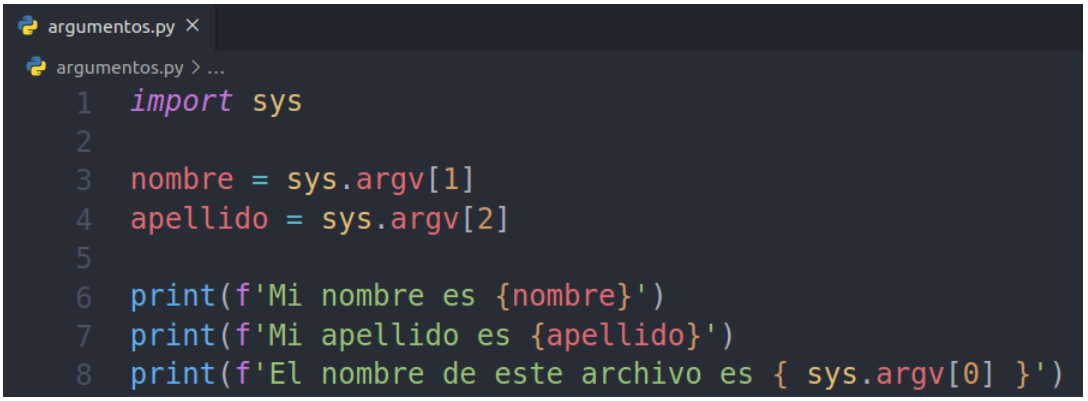

Imagen 1. Código final de argumentos.py
Fuente: Desafío Latam

5. Este script se ejecutará de la misma manera que cualquier script de Python, pero será necesario agregar los argumentos solicitados de la siguiente forma:

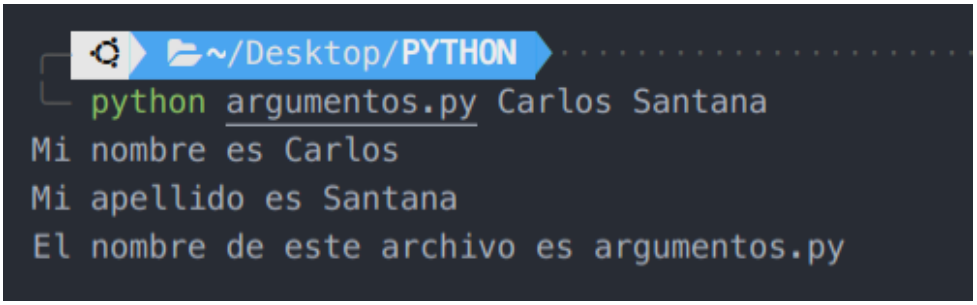

Imagen 2. Ejecutando argumentos.py en terminal
Fuente: Desafío Latam

Como se puede observar, las palabras Carlos y Santana que acompañan la ejecución del programa, se utilizarán como los argumentos `argv` en posiciones 1 y 2 respectivamente. Por otro lado, el argumento 0 siempre corresponderá al nombre de archivo en ejecución.

Si ejecutamos `type(sys.argv)` notaremos que se trata de una lista, esa es la razón por la que podemos utilizar índices para acceder a distintas partes de este. Es importante recordar que todas las operaciones y métodos que veamos para las listas se aplicarán también para el `sys.argv`.

```
import sys
type(sys.argv)
```

```
list
```

NOTA: El inconveniente que tiene `sys.argv` es que es poco intuitivo para el usuario ya que si no está familiarizado con el código, no podrá entender cada argumento inicialmente. Pero es fundamental cuando existen procesos automáticos agendados por el computador el cual ejecuta un archivo .py para realizar procesos claves para las empresas.

Operaciones y funciones básicas en una lista

Las listas de Python tienen una serie de funciones que permiten realizar operaciones básicas, entre ellas, ordenar los elementos de una lista, saber cuántas veces se encuentra un elemento allí, además de, borrar y agregar elementos. Por lo mismo, es muy importante saber leer la documentación asociada a éstas.

Leyendo la documentación de listas

La información oficial de Python sobre las listas (y de manera más general de estructuras de datos) se encuentra en docs.python.org. Se puede llegar fácilmente por medio de google, lo único que uno debe asegurarse es que la versión que se esté consultando sea la misma con la que se está trabajando.

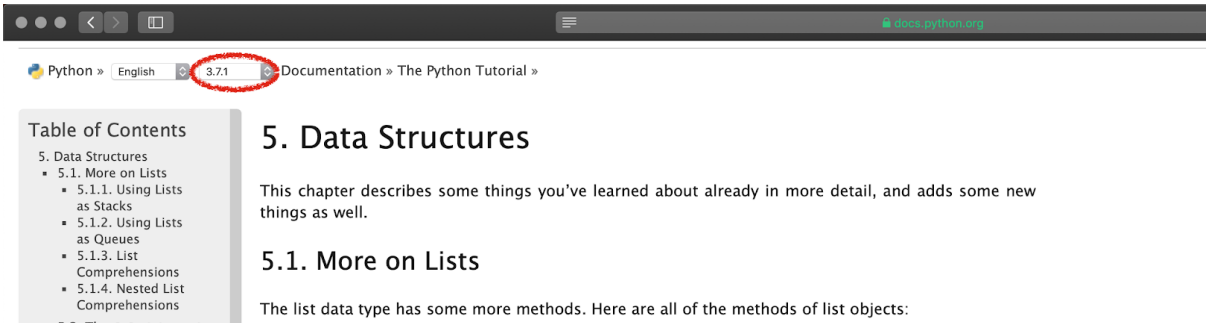

Imagen 3. Documentación Listas
Fuente: Python.org

Métodos aplicables a listas

Cuando definimos una lista en Python, el intérprete infiere cuál es la mejor representación de la expresión. En base a este punto, también le delega a la expresión una serie de acciones que pueda realizar, las que se conocen como **métodos de lista**.

Cuando generamos una nueva lista y la asignamos a una variable, su generación viene con una serie de atributos y métodos asociados. Esta es otra de las virtudes del paradigma orientado a objetos: cada objeto o representación creada vendrá con una serie de funcionalidades agregadas.

La forma tradicional para llamar a un método fue vista anteriormente y se llama notación de punto: `objeto.método(argumentos)`.

Al generar un objeto llamado `lista_de_numeros` que se compone de los números del 1 al 10, donde Python le concederá una serie de acciones dado que infiere que su mejor representación es una lista, donde se puede ver cuáles son todas las posibles acciones utilizando el atributo `__dir__().`

```
lista_de_numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9 ,10]
```

```
print(lista_de_numeros.__dir__())
```

```
['__repr__', '__hash__', '__getattribute__', '__lt__', '__le__',
'__eq__', '__ne__', '__gt__', '__ge__', '__iter__', '__init__',
'__len__', '__getitem__', '__setitem__', '__delitem__', '__add__',
'__mul__', '__rmul__', '__contains__', '__iadd__', '__imul__',
'__new__', '__reversed__', '__sizeof__', 'clear', 'copy', 'append',
'insert', 'extend', 'pop', 'remove', 'index', 'count', 'reverse',
'sort', '__doc__', '__str__', '__setattr__', '__delattr__',
'__reduce_ex__', '__reduce__', '__subclasshook__', '__init_subclass__',
'__format__', '__dir__', '__class__']
```

Aquellos elementos que están envueltos por `__` se conocen como **magic built-in o dunders**, y buscan generar flexibilizaciones en el comportamiento de las clases. Serán retomados cuando hablemos de clases. Todas las que no son dunders son las acciones disponibles a acceder. A continuación, revisaremos los más importantes:

Método `append(x)`

El método `append` agrega elementos al final de la lista.
Por ejemplo, si queremos agregar el celeste a nuestra lista de colores, se hace de la siguiente forma:

```
colores = ['verde', 'rojo', 'rosa', 'azul']
colores.append("celeste")
```

```
# pedimos una representación actualizada de la lista
print(colores)
```

```
['verde', 'rojo', 'rosa', 'azul', 'celeste']
```

Cabe destacar que no es necesario declarar de forma explícita la asignación al objeto, dado que `append` es una función inherente al objeto, opera y lo actualiza dentro de ella.

Método `insert(i, x)`

Este método nos permite agregar el elemento `x` en la posición `i` específica.

Por ejemplo, si agregamos el número 13 a nuestra `lista_de_numero`, tendremos el problema de que saltamos el número 12.

```
lista_de_numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
lista_de_numeros.append(13)
```

```
print(lista_de_numeros)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13]
```

Para resolver este problema, podemos especificar en qué posición de la lista vamos a incorporar un elemento. Para poder ingresar un elemento en una posición específica de la lista podemos hacer uso del método `.insert(i, x)`, donde la función necesita la posición a modificar como primer argumento `(i)` y el elemento a ingresar como el segundo `(x)`.

Insertemos el número 12 donde corresponde, la posición 11, ya que partimos desde cero.

```
lista_de_numeros.insert(11, 12)
```

```
print(lista_de_numeros)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

Método `pop()`

Para sacar el último elemento dentro de una lista, y obtenerlo, se debe utilizar el método `.pop()`. Por defecto, la función eliminará el último elemento de la lista y lo imprimirá. Si la expresión es asignada a una variable, la variable almacenará el elemento que se sacó.

También es posible pasar como argumento de la función una posición específica, de esta forma, se buscará dentro de la lista el elemento en esa posición, y este será eliminado y entregado.

```
colores.pop()
```

```
'celeste'
```

```
colores.pop(3)
```

```
'azul'
```

NOTA: Es bueno hacer notar que es posible almacenar el valor “popeado”:

```
color = colores.pop(0)
print(color)
```

```
'verde'
```

Método `remove(x)`

Para remover un elemento específico, se utiliza el método `remove(x)`, donde `x` es el elemento específico a eliminar. En caso que el elemento no esté presente en la lista, Python arrojará un error `ValueError`.

```
colores.remove("rojo")
```

```
print(colores)
```

```
['rosa']
```

```
# Se arroja un error dado que azul ya no es parte de colores
colores.remove("azul")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call
last)

<ipython-input-13-86badd94da6c> in <module>()
----> 1 colores.remove("azul")
```

```
ValueError: list.remove(x): x not in list
```

Método `reverse()`

Se puede invertir el orden de los elementos de una lista utilizando `.reverse`.

```
numeros = [100, 20, 70, 500]
animales = ["perro", "gato", "hurón", "erizo"]
numeros.reverse()
animales.reverse()
```

```
print(numeros)
```

```
[500, 70, 20, 100]
```

```
print(animales)
```

```
['erizo', 'hurón', 'gato', 'perro']
```

Método `sort()`

Se puede ordenar los elementos de forma ascendente utilizando `.sort`. En caso de que se trate de strings, se ordenan de forma ascendente en el abecedario.

```
animales.sort()
numeros.sort()
```

```
print(animales)
```

```
['erizo', 'gato', 'hurón', 'perro']
```

```
print(numeros)
```

```
[20, 70, 100, 500]
```

Si se quiere ordenar de forma descendente, se puede aplicar `reverse()` luego de aplicar `sort()`.

```
numeros.reverse()
print(numeros)
```

```
[500, 100, 70, 20]
```

Un aspecto relevante a recalcar y tomar en cuenta, es que al trabajar con listas en Python **todas estas funciones se realizaron sobre la misma lista**. Por tanto, si no se guarda la lista original en algún objeto separado, esta información se pierde.

Función `Sorted()`

Una alternativa al método sort es la función `sorted`. Esta obtiene los mismos resultados y en el caso de querer ordenar de manera descendente se utiliza el argumento `reverse = True`

```
# ordenamiento ascendente
sorted([3,6,7,4,1])
```

```
[1, 3, 4, 6, 7]
```

```
# ordenamiento descendente
sorted([3,6,7,4,1], reverse = True)
```

```
[7, 6, 4, 3, 1]
```

Método `index()`

El método index retornará el índice (de cero al que corresponda dependiendo del largo de la lista) en el cual se encuentra un elemento de interés, por ejemplo:

```
print(animales.index('gato'))
```

```
2
```

```
print(numeros.index(500))
```

```
0
```

Las strings poseen algunas propiedades muy similares a las listas. Básicamente es posible poder revisar sus elementos mediante el uso de índices, e incluso es posible leer en orden inverso utilizando triple índice.

Operaciones: Concatenación de listas

Como ya mostramos anteriormente, es importante entender una lista como **"una estructura de tipo lista"**, por lo tanto, hay algunos comportamientos que podrían sorprendernos, ya que no es lo que esperaríamos, donde un ejemplo de esto es la operación `+`.

Como vimos, los strings son un caso especial de listas, por lo tanto, las operaciones + e incluso `len()` funcionarán de manera análoga:

```
# definamos dos listas de animales
animales = ['Gato', 'Perro', 'Tortuga']
animales_2 = ['Hurón', 'Hamster', 'Erizo de Tierra']

# Si las concatenamos, podremos obtener una lista de mascotas
mascotas = animales + animales_2
# Veamos algunas características
print(animales)
print(len(animales))
print(animales_2)
print(len(animales_2))
print(mascotas)
print(len(mascotas))
```

```
['Gato', 'Perro', 'Tortuga']
3
['Hurón', 'Hamster', 'Erizo de Tierra']
3
['Gato', 'Perro', 'Tortuga', 'Hurón', 'Hamster', 'Erizo de Tierra']
6
```

Operaciones: repitiendo listas

Al utilizar el operador `*` con un string, se genera el mismo resultado que con los strings. Por ejemplo, se nos ha informado que hay 4 veces más perros, gatos y tortugas que hurones, hamsters y erizos de tierra, con estos datos podríamos generar una lista nueva e ingresar cada elemento nuevamente, pero sería tedioso para nosotros.

Entonces, lo que vamos a realizar es multiplicar los elementos contenidos en la lista mediante el operador `*`.

```
animales_actualizados = animales * 4

mascotas = animales_actualizados + animales_2

# Veamos algunas características
print(animales_actualizados)
print(len(animales_actualizados))
print(animales_2)
print(len(animales_2))
print(mascotas)
print(len(mascotas))
```

```
['Gato', 'Perro', 'Tortuga', 'Gato', 'Perro', 'Tortuga', 'Gato',
'Perro', 'Tortuga', 'Gato', 'Perro', 'Tortuga']
12
['Hurón', 'Hamster', 'Erizo de Tierra']
3
['Gato', 'Perro', 'Tortuga', 'Gato', 'Perro', 'Tortuga', 'Gato',
'Perro', 'Tortuga', 'Gato', 'Perro', 'Tortuga', 'Hurón', 'Hamster',
'Erizo de Tierra']
15
```