# Taller Integrador – Python.

# Unidad 2. Listas, estructuras y operadores.

## Tipo de dato LISTA.

El tipo lista en Python me permite almacenar en su interior, diferentes tipos de datos, separados por coma. El primer elemento de la lista ocupa la posición 0 (cero).

```
# Tipo de dato LISTA
lista = [1,2,3]
print(lista[0])
```

Si ejecutamos el código anterior, veremos por pantalla el elemento que ocupa la posición cero de la lista, en nuestro caso será el número 1.
Podemos cargar nuestra lista con elementos de varios tipos diferentes.

```
# Tipo de dato LISTA
lista = ['Hola',2,4.23]
print(lista[0])
```

Veremos por pantalla, al ejecutarlo, la palabra HOLA.
Dijimos que las listas aceptaban cualquier tipo de dato, por lo tanto, podemos cargarle otras listas, por ejemplo:

```
# Tipo de dato LISTA
lista_a = ['Hola']
lista_b = ['mundo']
lista = [lista_a[0],lista_b[0]]
print(lista[0]+' '+lista[1])
```

Veremos por pantalla, el famoso Hola mundo.

Podemos mostrar por pantalla todo el contenido de una lista sin necesidad de recorrerlo elemento por elemento, como vimos en el primer ejemplo o podemos manipular cada uno de los mismos, y recombinarlos en una nueva lista, utilizando algunos o todos los elementos de las listas originales.

## LEN.

La función LEN nos permite conocer la longitud de una lista.

```
# Tipo de dato LISTA
nombres = ['Juan','Pedro','Luis']
print(len(nombres))
```

El resultado será el número 3, ya que la lista tiene tres elementos.
Al igual que con los tipos de datos que vimos, también podemos ver con TYPE que tipo de dato es:

```
# Tipo de dato LISTA
nombres = ['Juan','Pedro','Luis']
print(type(nombres))
```

Esto veremos por pantalla:

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

## Métodos del tipo Lista.

- *APPEND*

Este método agrega un elemento al final de la lista.

```
# Tipo de dato LISTA (Métodos)
nombres = ['Juan','Pedro','Luis']
print(nombres)
nombres.append('Jose')
print(nombres)
```

En el ejemplo anterior, creamos una lista con los nombres Juan, Pedro y Luis.
Luego, utilizando el método append() agregamos, al final de la misma, el nombre Luis.
Si ejecutamos el código, esto es lo que veremos por pantalla:

```
================= RESTART: D:\0000_Clases\Python\practica.py ==
['Juan', 'Pedro', 'Luis']
['Juan', 'Pedro', 'Luis', 'Jose']
```

- *COUNT*

Este método recibe un elemento como argumento, y nos devuelve la cantidad de veces que aparece dicho elemento dentro de la lista.
Por ejemplo, si creamos una lista con algunas letras del abecedario, podremos contar la cantidad de veces que aparece cada una dentro de la misma:

```
# Tipo de dato LISTA (Métodos)
letras = ['a','b','b','c','d']
print('Cantidad de letra a:',letras.count('a'))
print('Cantidad de letra a:',letras.count('b'))
print('Cantidad de letra a:',letras.count('c'))
print('Cantidad de letra a:',letras.count('d'))
```

Y esto es lo que deberíamos ver en pantalla al ejecutar el código:

```
Cantidad de letra a: 1
Cantidad de letra a: 2
Cantidad de letra a: 1
Cantidad de letra a: 1
```

- _INDEX_

Este método recibe un elemento como argumento, y devuelve el índice de su primera aparición en la lista.
Por ejemplo, cargamos con números una lista de la siguiente manera, y mostramos el índice de la primera aparición de un elemento repetido:

```
# Tipo de dato LISTA (Métodos)
numeros = [1,2,3,5,6,3,7,9]
print(numeros.index(3))
```

Este es el resultado que veremos por pantalla:

```
2
```

Si el elemento no se encuentra en la lista, este método devolverá un mensaje de error.

```
Traceback (most recent call last):
  File "D:\0000_Clases\Python\practica.py", line 3, in <module>
    print(numeros.index(12))
ValueError: 12 is not in list
```

- _INSERT_

Permite insertar un elemento en la lista, en el índice que necesitemos.
Por ejemplo, tenemos la lista de letras a, b y d pero queremos agregar la letra c inmediatamente después de la letra b.

```
# Tipo de dato LISTA (Métodos)
letras = ['a','b','d']
print(letras)
letras.insert(2,'c')
print(letras)
```

Al ejecutarlo, esto es lo que veremos por pantalla:

```
['a', 'b', 'd']
['a', 'b', 'c', 'd']
```

- _POP_

Este método devuelve el último elemento de una lista, y lo borra.
Por ejemplo, en los envíos de datos en los que el último elemento es un código de verificación para comprobar que los datos han sido transferidos por completo y sin errores.

```
# Tipo de dato LISTA (Métodos)
numeros = [1,2,3,4,5,6]
print(numeros)
numeros.pop()
print(numeros)
```

Esto es lo que veremos por pantalla ejecutando el ejemplo anterior:

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

- _REMOVE_

Este método recibe como argumento un elemento y borra su primera aparición en la lista.
En caso de no encontrar al elemento en la lista, devuelve un mensaje de error.
Veamos un ejemplo en el que cargamos una lista con los números del 1 al 6, los mostramos por pantalla y luego eliminamos el número 1.

```
# Tipo de dato LISTA (Métodos)
numeros = [1,2,3,4,5,6]
print(numeros)
numeros.remove(1)
print(numeros)
```

Así veremos el resultado de la ejecución por pantalla:

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

- *REVERSE*

Invierte el orden de los elementos de una lista.

```
# Tipo de dato LISTA (Métodos)
letras = ['a','b','c','d']
print(letras)
letras.reverse()
print(letras)
```

Mostramos por pantalla la lista original y la lista invertida:

```
['a', 'b', 'c', 'd']
['d', 'c', 'b', 'a']
```

- *SORT*

Ordena los elementos de una lista.

```
# Tipo de dato LISTA (Métodos)
numeros = [2,8,1,13,9,0,5]
print(numeros)
numeros.sort()
print(numeros)
```

Este es el resultado visto por pantalla:

```
[2, 8, 1, 13, 9, 0, 5]
[0, 1, 2, 5, 8, 9, 13]
```

También tenemos la opción de ordenarlos en sentido inverso, de la siguiente manera:

```
# Tipo de dato LISTA (Métodos)
numeros = [2,8,1,13,9,0,5]
print(numeros)
numeros.sort(reverse=True)
print(numeros)
```

Al poner el parámetro reverse en True, el ordenamiento se hace de mayor a menor:

```
[2, 8, 1, 13, 9, 0, 5]
[13, 9, 8, 5, 2, 1, 0]
```

# Estructuras.

## Condicional IF

El condicional if se utiliza básicamente para tomar decisiones.
Esta decisión está basada en una operación lógica (verdadera o falsa).
Veamos un ejemplo con ingreso de números por teclado, pero vamos a ver como convertimos los datos tipo texto que toma por defecto el input en un entero.

```
# Conversión de datos en input
na = int(input('numero 1:'))
nb = int(input('numero 2:'))
print(na+nb)
```

La sintaxis de la estructura de decisión es la siguiente:

```
# Estructura de decisión
if condición :
    código a ejecutar si la condición es verdadera
elif otra condición:
    código a ejecutar si la otra condición es verdadera.
else:
    código a ejecutar en caso de no existir condición verdadera.
```

Por ejemplo, le pedimos al usuario que ingrese un valor entero, mayor que cero y verificamos si el número es par o impar.

```
# Par o Impar
numero = int(input('Ingrese un número entero mayor que cero: '))
resto = numero%2
if resto == 0:
    print('El número ingresado es PAR')
else:
    print('El número ingresado es IMPAR')
```

IMPORTANTE: hay que respetar el identado del código a ejecutar dentro del if, elif y else como así también los dos puntos que lleva cada uno.

```
================ RESTART: D:\0000_Clases\Python\practica.py ================
Ingrese un número entero mayor que cero: 123
El número ingresado es IMPAR
>>>
================ RESTART: D:\0000_Clases\Python\practica.py ================
Ingrese un número entero mayor que cero: 125678
El número ingresado es PAR
>>>
```

## Operadores relacionales.

Son aquellos que utilizamos para evaluar determinada condición.

| Símbolo | Significado | Ejemplo | Resultado |
|---|---|---|---|
| `==` | Igual que | `5 == 7` | `False` |
| `!=` | Distinto que | `rojo != verde` | `True` |
| `<` | Menor que | `8 < 12` | `True` |
| `>` | Mayor que | `12 > 7` | `True` |
| `<=` | Menor o igual que | `12 <= 12` | `True` |
| `>=` | Mayor o igual que | `4 >= 5` | `False` |

## Operadores lógicos.

| Operador | Ejemplo | Explicación | Resultado |
|---|---|---|---|
| `and` | `5 == 7 and 7 < 12` | `False and False` | `False` |
| `and` | `9 < 12 and 12 > 7` | `True and True` | `True` |
| `and` | `9 < 12 and 12 > 15` | `True and False` | `False` |
| `or` | `12 == 12 or 15 < 7` | `True or False` | `True` |
| `or` | `7 > 5 or 9 < 12` | `True or True` | `True` |
| `xor` | `4 == 4 xor 9 > 3` | `True o True` | `False` |
| `xor` | `4 == 4 xor 9 < 3` | `True o False` | `True` |

## Estructura While.

Es una estructura de repetición, que me permite ejecutar una porción de código la cantidad de veces que le indiquemos.

Estas repeticiones son controladas por el programador, por ejemplo, mediante el uso de un contador o el resultado de un evento.

El siguiente ejemplo, es una estructura While controlada por un contador:

```
# Estructura While por contador
contador = 1
while contador <=10:
    print(contador)
    contador=contador+1
print('Fin del ciclo')
```

Al ejecutarlo, este es el resultado que deberíamos ver por pantalla:

```
1
2
3
4
5
6
7
8
9
10
Fin del ciclo
```

En otros lenguajes tenemos dos tipos de ciclos, el While y el Do While.
En Python (al menos en la versión 3) tenemos solo el ciclo While. Está en nosotros, como programadores, armar la estructura para que se comporte de una u otra manera.
Recordemos que la diferencia entre una y otra estructura es que el While evalúa la condición antes de empezar el ciclo y el Do While ejecuta al menos una vez el ciclo y recién después empieza a evaluar la condición.

Ejemplo While:

```
# Ejemplo While
contador = 1
while contador <=10:
    print(contador)
    contador=contador+1
print('Fin del ciclo')
```

Ejemplo Do While:

```
# Ejemplo Do While
contador = 0
while True:
    contador=contador+1
    print(contador)
    if contador==11:
        break
print('Fin del ciclo')
```

Veamos ahora una estructura While controlada por un evento.
Le pedimos al usuario que ingrese una serie de notas del 1 al 10 y cuando no quiera ingresar más valores, que ingrese un -1.
Una vez ingresado el -1, dicho evento termina el ciclo y continúa con la ejecución del programa, mostrando los valores ingresados, la suma y el promedio.

```
# Ejemplo While por evento.
notas=[]
nota=0
while nota!=-1:
    nota=int(input('Ingrese la nota de 1 a 10 o -1 para terminar: '))
    if nota==-1:
        break
    notas.append(nota)
print('Notas ingresadas: ',notas)
print('Suma de notas: ',sum(notas))
print('Promedio de las notas: ',sum(notas)/len(notas))
```

Incorporamos el método sum que nos proporciona la suma de los elementos numéricos de una lista.
Esto veríamos por pantalla al ejecutar el código e ingresarlo, por ejemplo, los valores 8 y 7:

```
Ingrese la nota de 1 a 10 o -1 para terminar: 8
Ingrese la nota de 1 a 10 o -1 para terminar: 7
Ingrese la nota de 1 a 10 o -1 para terminar: -1
Notas ingresadas:  [8, 7]
Suma de notas:  15
Promedio de las notas:  7.5
```

## Estructura FOR.

En Python, la sentencia for tiene un uso que difiere un poco del que se le da en otros lenguajes.
En nuestro caso la sentencia for itera sobre los ítems de cualquier secuencia, por ejemplo, una lista, en el orden en el que aparecen los elementos.
Por ejemplo:

```
# Ejemplo For.
numeros = [1,2,3,4,5,6]
for elemento in numeros:
    print (elemento)
```

El ciclo for recorre toda la lista.
También podemos usar esta estructura como generadora de elementos para utilizarlos en otra parte de nuestro código mediante el atributo range().

```
# Ejemplo For con rango
for año in range(2000,2005):
    print (año)
```

En este ejemplo le pedimos que muestre los años comprendidos entre el 2000 y el 2005.
Si prestamos atención a lo que se ve por pantalla luego de la ejecución, notaremos que la función, toma inicia en el primer valor del rango y termina uno antes del final.

```
2000
2001
2002
2003
2004
```

Este detalle es importante para no obtener un resultado inesperado.

Lo interesante de Python es que nos permite componer nuestro código combinando las diferentes sentencias, métodos y funciones.
Esto facilita muchísimo el desarrollo, acorta los tiempos y nos permite generar un código simple y fácil de mantener.

A la función range() le podemos agregar un tercer valor, para indicarle el incremento o decremento que queremos obtener.

```
# Ejemplo For con incremento
for numero in range(0,10,2):
    print (numero)
```

Este es el resultado visto por pantalla:

```
0
2
4
6
8
```

## Cadenas de caracteres.

Vimos que además de números, Python es capaz de manipular cadenas de caracteres.
Estas pueden estar encerradas entre comillas simples o dobles, y el carácter \ se utiliza para escapar a as comillas.

```
print("Texto sin comillas")
print ("Texto con \' una comilla simple")
```

```
Texto sin comillas
Texto con ' una comilla simple
```

Si no queremos que los caracteres de escape se interpreten como tal, utilizamos la cadena cruda, anteponiendo una letra r antes de abrir las comillas.

```
print("Texto sin comillas")
print (r"Texto con \' una comilla simple")
```

```
Texto sin comillas
Texto con \' una comilla simple
```

Las cadenas de texto pueden concatenarse con el operador + y pueden repetirse con el operador *.

```
print('Hola' + 'Mundo')
print (3 * 'Hola ')
```

```
HolaMundo
Hola Hola Hola
```

También serán concatenadas dos o más cadenas que, sin estar ligadas con el operador +, estén juntas.

```
print('Hola' 'Mundo')
```

```
HolaMundo
```

Las cadenas de texto se pueden indexar. El primer carácter de dicha cadena tendrá el índice 0 (cero).
No existe en Python un tipo especial para los caracteres, simplemente se los considera como una cadena de longitud 1.

```
cadena = 'Taller Integrador'
print(cadena[0])
```

Veremos por pantalla la letra T, que es el carácter que ocupa el índice cero dentro de la cadena de texto.

Si necesitamos empezar a utilizar los caracteres de una cadena de atrás hacia adelante, podemos averiguar su longitud con la función LEN o podemos utilizar argumentos negativos, siendo el -1 el último elemento de la cadena, -2 el penúltimo y así sucesivamente.

```
cadena = 'Taller Integrador'
print(cadena[-1])
```

Al ejecutar el ejemplo anterior, veremos por pantalla la letra r.

## Sub cadenas.

En Python tenemos disponible el concepto de sub cadenas o rebanadas como figura en muchos documentos del lenguaje.
La notación es simple cadena [inicio: final]
Al igual que en la función RANGE, el inicio está incluido en la sub cadena, pero no el final.
Por ejemplo, si queremos obtener, de nuestra cadena "Taller Integrador" solo la palabra "Integrador", hacemos lo siguiente:

```
palabra = 'Taller Integrador'
print(palabra[7:17])
```

Esto es lo que veremos por pantalla:

```
Integrador
```

Al obtener una sub cadena, no estamos modificando en nada a la cadena original, solo estamos mostrando una parte de esta o, en el caso de asignarla a otra variable, estaremos creando una nueva cadena de texto.

```
palabra = 'Taller Integrador'
palabra_1 = palabra[7:17]
palabra_2 = palabra[0:7]

print(palabra_1 +' '+ palabra_2)
```

Esto vemos por pantalla:

```
Integrador Taller
```

Los parámetros de inicio y fin de las sub cadenas, tienen valores por defecto.
En el caso del inicio, si no lo aclaramos, tomara por defecto el primer elemento de la cadena de texto.
En el caso del final, si no se lo indicamos, tomará por defecto al último elemento de nuestra cadena.

```
palabra = 'Taller Integrador'

print(palabra[:7])

print (palabra[2:])
```

En el primer PRINT lo que intentamos hacer es mostrar por pantalla, los elementos contenidos dentro de la variable "palabra" comprendidos entre el primero y el sexto elemento.
El cambio, en el segundo PRINT, mostramos desde el segundo elemento hasta el último.
Esto veremos por pantalla:

```
Taller
ller Integrador
```

Las cadenas en Python son inmutables, no pueden ser modificadas.
Por ejemplo, si intentamos agregar un nuevo elemento a una cadena ya existente, el intérprete nos dará un mensaje de error.

```
palabra = 'abc'
palabra[0]='z'
```

Si ejecutamos lo anterior, veremos el siguiente mensaje de error:

```
Traceback (most recent call last):
  File "D:\0000_Clases\Python\practica.py", line 2, in <module>
    palabra[0]='z'
TypeError: 'str' object does not support item assignment
```