0.1 Tercer Modulo- Python , Data Types

Erick Mejia

1. Comentarios

1.0.1 ¿Qué son?

Son textos escritos en archivos de Python que el programa ignora, es decir, no se ejecutan.

1.0.2 ¿Cuál es su utilidad?

- Ayudan a explicar el código y hacerlo más fácil de entender.
- Lo ideal es escribir código claro que no necesite muchos comentarios para ser comprendido.

1.0.3 Tipos de comentarios

Comentarios de una línea:

- Se escriben usando el símbolo "#".
- Se usan para explicar partes simples del código.

In [73]:
```
# Esto es una instrucción print
print('Hello world') # Esto es una instrucción print
```

```
Hello world
```

Comentarios de varias líneas:

- Texto encapsulado en triples comillas (que pueden ser tanto comillas simples como dobles).
- Se suele usar para documentar bloques de código más significativos.

In [74]:
```
def producto(x, y):
    '''
    Esta función recibe dos números como parámetros y devuelve
    como resultado el producto de los mismos.
    '''
    return x * y
```

2. Literales, variables y tipos de datos básicos

De forma muy genérica, al ejecutarse un programa Python, simplemente se realizan operaciones sobre objetos. Estos dos términos son fundamentales.

- Objetos: cualquier tipo de datos (números, caracteres o datos más complejos)
- Operaciones: cómo manipulamos estos datos.

Ejemplo:

In [75]:
```
4 + 3
```

Out[75]:
```
7
```

2.1 Literales

- Python tiene varios tipos de datos incluidos en el lenguaje.
- Los literales son formas directas de escribir esos datos en el código.
- Dependiendo del tipo, los datos pueden ser:
  - Simples o compuestos.
  - Mutables o immutables

Literales simples

- Enteros
- Decimales
- Flotantes
- Booleanos

In [76]:
```
print(4)                            # número entero
print(4.2)                          # número en coma flotante
print('Hello world!')               # string
print(False)
```

```
4
4.2
Hello world!
False
```

Literales compuestos

- Tuplas
- Listas
- Diccionarios
- Conjuntos

In [77]:
```
print([1, 2, 3, 3])                                 # lista - mutable
print({'Nombre' : 'John Doe', "edad": 30})          # Diccionario - mutable
print({1, 2, 3, 3})                                 # Conjunto - mutable
print((4, 5))                                       # tupla - inmutable
2, 4                                                # tupla
```

```
[1, 2, 3, 3]
{'Nombre': 'John Doe', 'edad': 30}
{1, 2, 3}
(4, 5)
```

Out[77]:
```
(2, 4)
```

2.2 Variables

- Las variables sirven para apuntar o referirse a un valor.
- Las variables y los valores se guardan en lugares distintos de la memoria.
- Una variable siempre apunta a un valor, no a otra variable.
- Algunos valores pueden contener otros valores, como las listas.
- Se usan sentencias de asignación para dar un valor a una variable.

`<nombre_variable> '=' <objeto>`

In [78]:
```
# # Asignación de variables
a = 5
print(a)
```

```
5
```

In [79]:
```
a = 1                   # entero
b = 4.0                 # coma flotante
c = "ITQ"               # string
d = 10 + 1j             # numero complejo
e = True                #False # boolean
f = None                # None

# visualizar valor de las variables y su tipo
print(a)
print(type(a))

print(b)
```

```
print(type(b))

print(c)
print(type(c))

print(d)
print(type(d))

print(e)
print(type(e))

print(f)
print(type(f))
```

```
1
<class 'int'>
4.0
<class 'float'>
ITQ
<class 'str'>
(10+1j)
<class 'complex'>
True
<class 'bool'>
None
<class 'NoneType'>
```

- Las variables no tienen tipo.
- Las variables apuntan a objetos que sí lo tienen
- Dado que Python es un lenguaje de tipado dinámico, la misma variable puede apuntar, en momentos diferentes de la ejecución del programa, a objetos de diferente tipo.

In [80]:
```
a = 3
print(a)
print(type(a))

a = 'Pablo García'
print(a)
print(type(a))

a = 4.5
print(a)
print(type(a))
```

```
3
<class 'int'>
Pablo García
<class 'str'>
4.5
<class 'float'>
```

- Garbage collection: Cuando un objeto deja de estar referenciado, se elimina automáticamente.

Literales compuestos

- Podemos obtener un identificador único para los objetos referenciados por variables.
- Este identificador se obtiene a partir de la dirección de memoria.

In [81]:
```
a = 3
print(id(a))

a = 'Pablo García'
print(id(a))

a = 4.5
print(id(a))
```

```
140711968494440
1631146958960
1631139239696
```

- Referencias compartidas: un mismo objeto puede ser referenciado por más de una variable.
  - Variables que referencian al mismo objeto tienen mismo identificador.

In [82]:
```
a = 4567
print(id(a))
```

```
1631149062128
```

In [83]:
```
b = a
print(id(b))
```

```
1631149062128
```

In [84]:
```
c = 4567
print(id(c))
```

```
1631149059792
```

In [85]:
```
a = 25
b = 25

print(id(a))
print(id(b))
print(id(25))
```

```
140711968495144
140711968495144
140711968495144
```

In [86]:
```
# # Ojo con los enteros "grandes" [-5, 256]
a = 258
b = 258

print(id(a))
print(id(b))
print(id(258))
```

```
1631149061584
1631149059984
1631149061808
```

- Referencia al mismo objeto a través de asignar una variable a otra.

In [87]:
```
a = 400
b = a
print(id(a))
print(id(b))
```

```
1631149061264
1631149061264
```

- Las variables pueden aparecer en expresiones.

In [88]:
```
a = 3
b = 5
print(id(a))
print (a + b)
```

```
140711968494440
8
```

In [89]:
```
c = a + b
print(c)
print(id(c))
```

```
8
140711968494600
```

Respecto a los nombres de las variables

- No se pueden empezar con números.
- Por costumbre, es mejor usar snake_case, es decir, separar palabras con guion bajo _.
- Python distingue entre mayúsculas y minúsculas.
- Los nombres deben ser claros y describir lo que guardan.
- Existen palabras reservadas del lenguaje.
- Hay que tener cuidado de no usar esos nombres para variables.

In [90]:
```
print(pow(3,2))
```

```
9
```

In [91]:
```
print(pow(3,2))

pow = 1 # built-in reasignado
print(pow)

print(pow(3,2))
```

```
9
1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[91], line 6
      3 pow = 1 # built-in reasignado
      4 print(pow)
----> 6 print(pow(3,2))

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

In [92]:
```
def pow(a, b):
    return a + b
print(pow(3,2))
```

```
5
```

Asignación múltiple de variables

In [93]:
```
x, y, z = 1, 2, 3
print(x, y, z)

t = x, y, z, 7, "Python"
print(t)
print(type(t))
```

```
1 2 3
(1, 2, 3, 7, 'Python')
<class 'tuple'>
```

- Esta técnica tiene un uso interesante: el intercambio de valores entre dos variables.

In [94]:
```
a = 1
b = 2
a, b = b, a
print(a, b)
```

```
2 1
```

In [95]:
```
a = 1
b = 2

c = a
a = b
b = c
print(a, b, c)
```

```
2 1 1
```

2.3 Tipos de datos básicos

Bool

- 2 posibles valores: 'True' o 'False'.

In [96]:
```
a = False
b = True
```

```
print(a)
print(type(a))

print(b)
print(type(b))
```

```
False
<class 'bool'>
True
<class 'bool'>
```

- 'True' y 'False' también son objetos que se guardan en caché, al igual que los enteros pequeños.

In [97]:
```
a = True
b = False

print(id(a))
print(id(b))

print(a is b)
print(a == b)
```

```
140711967025728
140711967025760
False
False
```

- Diferentes representaciones: base 10, 2, 8, 16.

In [98]:
```
x = 58              # decimal
z = 0b00111010      # binario
w = 0o72            # octal
y = 0x3A            # hexadecimal

print(x == y == z == w)
```

```
True
```

Strings

- Cadenas de caracteres.
- Son secuencias: la posición de los caracteres es importante.
- Son immutables: las operaciones sobre strings no cambian el string original.

In [99]:
```
s = 'John "ee" Doe'
print(s[0])           # Primer carácter del string.
print(s[-1])          # Último carácter del string.
print(s[1:8:2])       # Substring desde el segundo carácter (inclusive) hasta el␣↪
print(s[:])           # Todo el string.
print(s + "e")        # Concatenación.
```

```
J
e
on"e
John "ee" Doe
John "ee" Doee
```

2.4 Conversión entre tipos

- A veces queremos que un objeto sea de un tipo específico.
- Podemos obtener objetos de un tipo a partir de objetos de un tipo diferente (casting).

In [100…

```python
a = int(2.8)          # a será 2
b = int("3")          # b será 3
c = float(1)          # c será 1.0
d = float("3")        # d será 3.0
e = str(2)            # e será '2'
f = str(3.0)          # f será '3.0'
g = bool("a")         # g será True
h = bool("")          # h será False
i = bool(3)           # i será True
j = bool(0)           # j será False
k = bool(None)

print(a)
print(type(a))
print(b)
print(type(b))
print(c)
print(type(c))
print(d)
print(type(d))
print(e)
print(type(e))
print(f)
print(type(f))
print(g)
print(type(g))
print(h)
print(type(h))
print(i)
print(type(i))
print(j)
print(type(j))
print(k)
```

```
2
<class 'int'>
3
<class 'int'>
1.0
<class 'float'>
3.0
<class 'float'>
2
<class 'str'>
3.0
<class 'str'>
True
<class 'bool'>
False
<class 'bool'>
True
<class 'bool'>
False
<class 'bool'>
False
```

In [101…

```
print(7/4)       # División convencional. Resultado de tipo 'float'
print(7//4)      # División entera. Resultado de tipo 'int'
print(int(7/4))  # Divisón convencional. Conversión del resultado de 'float' a␣↪'ir
```

```
1.75
1
1
```

2.5 Operadores

- Combinación de valores, variables y operadores
- Operadores y operandos

Operadores aritméticos

Operador	Desc
a + b	Suma
a - b	Resta
a / b	División
a // b	División Entera
a % b	Modulo / Resto
a * b	Multiplicacion
a ** b	Exponenciación

In [102…

```
x = 3
y = 2

print('x + y = ', x + y)
print('x - y = ', x - y)
print('x * y = ', x * y)
```

```
print('x / y = ', x / y)
print('x // y = ', x // y)
print('x % y = ', x % y)
print('x ** y = ', x ** y)
```

```
x + y =  5
x - y =  1
x * y =  6
x / y =  1.5
x // y =  1
x % y =  1
x ** y =  9
```

Operadores de comparación

Operador	Desc
a > b	Mayor
a < b	Menor
a == b	Igualdad
a != b	Desigualdad
a >= b	Mayor o Igual
a <= b	Menor o Igual

In [103…

```
x = 10
y = 12

print('x > y es ', x > y)
print('x < y es ', x < y)
print('x == y es ', x == y)
print('x != y es ', x != y)
print('x >= y es ', x >= y)
print('x <= y es ', x <= y)
```

```
x > y es  False
x < y es  True
x == y es  False
x != y es  True
x >= y es  False
x <= y es  True
```

Operadores Lógicos

Operador	Desc
a and b	True, si ambos son True
a or b	True, si alguno de los dos es True
a ^ b	XOR - True, si solo uno de los dos es True
not a	Negación

In [104…

```
x = True
y = False
print('x and y es :', x and y)
print('x or y es :', x or y)
```

```
print('x xor y es :', x ^ y)
print('not x es :', not x)
```

```
x and y es : False
x or y es : True
x xor y es : True
not x es : False
```

Operadores Bitwise / Binarios

Operador	Desc
a & b	And binario
a | b	Or binario
a ^ b	Xor binario
~ a	Not binario
a » b	Desplazamiento binario a derecha
a « b	Desplazamiento binario a izquierda

In [105…

```
# x = 0b01100110
# y = 0b00110011
# print("Not x = " + bin(~x))
# print("x and y = " + bin(x & y))
# print("x or y = " + bin(x | y))
# print("x xor y = " + bin(x ^ y))
# print("x << 2 = " + bin(x << 2))
# print("x >> 2 = " + bin(x >> 2))
```

Operadores de Asignación

Operador	Desc
=	Asignación
+=	Suma y asignación
-=	Resta y asignación
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
//=	División entera y asignación
**=	Exponencial y asignación
&=	And y asignación
|=	Or y asignación
^=	Xor y asignación
»=	Despl. Derecha y asignación
«=	DEspl. Izquierda y asignación

In [106…

```
a = 5
a *= 3          # a = a * 3
a += 1          # No existe a++, ni ++a,a--,--a
print(a)
b = 6
```

```
b -= 2          # b = b - 2
print(b)
```

```
16
4
```

Operadores de Identidad

Operador	Desc
a is b	True, si ambos operadores son una referencia al mismo objeto
a is not b	True, si ambos operadores *no* son una referencia al mismo objeto

In [107…

```
a = 4444
b = a
print(a is b)
print(a is not b)
```

```
True
False
```

Operadores de Pertenencia

Operador	Desc
a in b	True, si a se encuentra en la secuencia b
a not in b	True, si a no se encuentra en la secuencia b

In [108…

```
x = 'Hola Mundo'
y = {1:'a',2:'b'}

print('H' in x) # True
print('hola' not in x) # True

print(1 in y) # True
print('a' in y) # False
```

```
True
True
True
False
```

2.6 Entrada de valores

In [109…

```
valor = input("Inserte valor:")
print(valor)
print(type(valor))
```

```
3
<class 'str'>
```

In [110…
```
grados_c = int(input("Conversión de grados a fahrenheit, inserte un valor: "))
print(f"Grados F: {1.8 * (grados_c) + 32}")
```

```
Grados F: 37.4
```

3. Tipos de datos compuestos (colecciones)

3.1 Listas

- Una colección de objetos.
- Mutables.
- Tipos arbitrarios heterogeneos.
- Puede contener duplicados.
- No tienen tamaño fijo. Pueden contener tantos elementos como quepan en memoria.
- Los elementos van ordenados por posición.
- Se acceden usando la sintaxis: [index].
- Los índices van de 0 a n-1, donde n es el número de elementos de la lista.
- Son un tipo de Secuencia, al igual que los strings; por lo tanto, el orden (es decir, la posición de los objetos de la lista) es importante.
- Soportan anidamiento.
- Son una implementación del tipo abstracto de datos: Array Dinámico.

Operaciones con listas

- Creación de listas.

In [111…
```
letras = ['a', 'b', 'c', 'd']
palabras = 'Hola mundo como estas'.split()
numeros = list(range(7))

print(letras)
print(palabras)
print(numeros)
print(type(numeros))
```

```
['a', 'b', 'c', 'd']
['Hola', 'mundo', 'como', 'estas']
[0, 1, 2, 3, 4, 5, 6]
<class 'list'>
```

In [112…
```
# # Pueden contener elementos arbitrarios / heterogeneos
mezcla = [1, 3.4, 'a', None, False]
print(mezcla)
print(len(mezcla)) # len me da el tamaño de la lista número de elementos
```

```
[1, 3.4, 'a', None, False]
5
```

In [113…
```
 # # Pueden incluso contener objetos más "complejos"
lista_con_funcion = [1, 2, len, pow]
```

```
print(lista_con_funcion)
```

```
[1, 2, <built-in function len>, <function pow at 0x0000017BC89F6F20>]
```

In [114…
```
# # Pueden contener duplicados
lista_con_duplicados = [1, 2, 3, 3, 3, 4]
print(lista_con_duplicados)
```

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

- Obtención de la longitud de una lista.

In [115…
```
letras = ['a', 'b', 'c', 'd', 1]
print(len(letras))
```

```
5
```

- Acceso a un elemento de una lista

In [116…
```
print(letras[2])
print(letras[-5])
print(letras[0])
```

```
c
a
a
```

- Slicing: obtención de un fragmento de una lista, devuelve una copia de una parte de la lista
  - Sintaxis: lista [inicio : fin : paso]

In [117…
```
letras = ['a', 'b', 'c', 'd', 'e']

print(letras[1:3])
print(letras[:3])
print(letras[:-1])
print(letras[2:])
print(letras[:])
print(letras[::2])
```

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

In [118…
```
letras = ['a', 'b', 'c', 'd']

print(letras)
print(id(letras))

a = letras[:]
print(a)
```

```
print(id(a))

print(letras.copy())
print(id(letras.copy()))
```

```
['a', 'b', 'c', 'd']
1631153248384
['a', 'b', 'c', 'd']
1631153247936
['a', 'b', 'c', 'd']
1631153013120
```

- Añadir un elemento al final de la lista

letras.append('e') print(letras) print(id(letras))

In [119…
```
letras += 'e'
print(letras)
print(id(letras))
```

```
['a', 'b', 'c', 'd', 'e']
1631153248384
```

- Insertar en posición.

In [120…
```
print(len(letras))
letras.insert(1,'g')
print(len(letras))
print(letras)
print(id(letras))
```

```
5
6
['a', 'g', 'b', 'c', 'd', 'e']
1631153248384
```

- Modificación de la lista (individual).

In [121…
```
letras[5] = 'f'
print(letras)
print(id(letras))
```

```
['a', 'g', 'b', 'c', 'd', 'f']
1631153248384
```

In [122…
```
# # index tiene que estar en rango
letras[20] = 'r'
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[122], line 2
      1 # # index tiene que estar en rango
----> 2 letras[20] = 'r'

IndexError: list assignment index out of range
```

- Modificación múltiple usando slicing.

In [123…
```
letras = ['a', 'b', 'c', 'f', 'g', 'h', 'i', 'j', 'k']
print(id(letras))
```

```
1631155044544
```

In [124…
```
letras[0:7:2] = ['z', 'x', 'y','p']
print(letras)
# print(id(letras))
```

```
['z', 'b', 'x', 'f', 'y', 'h', 'p', 'j', 'k']
```

In [125…
```
 # # Ojo con la diferencia entre modificación individual y múltiple. Asignación␣↪ir
numeros = [1, 2, 3]
numeros[1] = [10, 20, 30]

print(numeros)
print(numeros[1][0])

numeros[1][2] = [100, 200]
print(numeros)

print(numeros[1][2][1])
```

```
[1, [10, 20, 30], 3]
10
[1, [10, 20, [100, 200]], 3]
200
```

- Eliminar un elemento.

In [126…
```
#letras.remove('f')
if 'p' in letras:
    letras.remove('p')
#      #letras.remove('z')
print(letras)
```

```
['z', 'b', 'x', 'f', 'y', 'h', 'j', 'k']
```

In [127…
```
# elimina el elemento en posición -1 y lo devuelve
elemento = letras.pop()
print(elemento)
print(letras)
```

```
k
['z', 'b', 'x', 'f', 'y', 'h', 'j']
```

In [128…
```
numeros = [1, 2, 3]
print(numeros)
numeros[2] = [10, 20, 30]
print(numeros)
n = numeros[2].pop()
print(numeros)
print(n)
```

```
[1, 2, 3]
[1, 2, [10, 20, 30]]
[1, 2, [10, 20]]
30
```

In [129…
```
# numeros1=10
# print(numeros1)
# print(numeros)
```

In [130…
```
lista = []
a = lista.pop()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[130], line 2
      1 lista = []
----> 2 a = lista.pop()

IndexError: pop from empty list
```

- Encontrar índice de un elemento.

In [131…
```
letras = ['a', 'b', 'c', 'c']
if 'a' in letras:
    print(letras.index('a'))
print(letras)
```

```
0
['a', 'b', 'c', 'c']
```

- Concatenar listas.

In [132…
```
lacteos = ['queso', 'leche']
frutas = ['naranja', 'manzana']
print(id(lacteos))
print(id(frutas))

compra = lacteos + frutas
print(id(compra))
print(compra)
```

```
1631151799488
1631151717056
1631151708160
['queso', 'leche', 'naranja', 'manzana']
```

In [133…
```
# # Concatenación sin crear una nueva lista
frutas = ['naranja', 'manzana']
print(id(frutas))
frutas.extend(['pera', 'uvas'])
print(frutas)
print(id(frutas))
```

```
1631151712768
['naranja', 'manzana', 'pera', 'uvas']
1631151712768
```

In [134…
```
# # Anidar sin crear una nueva lista
frutas = ['naranja', 'manzana']
print(id(frutas))
frutas.append(['pera', 'uvas'])
print(frutas)
print(id(frutas))
```

```
1631153056448
['naranja', 'manzana', ['pera', 'uvas']]
1631153056448
```

- Replicar una lista.

In [135…
```
lacteos = ['queso', 'leche']
print(lacteos * 3)
print(id(lacteos))

a = 3 * lacteos
print(a)
print(id(a))
```

```
['queso', 'leche', 'queso', 'leche', 'queso', 'leche']
1631151705856
['queso', 'leche', 'queso', 'leche', 'queso', 'leche']
1631151705216
```

- Copiar una lista

In [136…
```
frutas2 = frutas.copy()
frutas2 = frutas[:]
print(frutas2)
print('id frutas = ' + str(id(frutas)))
print('id frutas2 = ' + str(id(frutas2)))
```

```
['naranja', 'manzana', ['pera', 'uvas']]
id frutas = 1631153056448
id frutas2 = 1631151710080
```

- Ordenar una lista.

In [137…
```
lista = [4,3,8,1]
print(lista)
lista.sort()
```

```
print(lista)

lista.sort(reverse=True)
print(lista)
```

```
[4, 3, 8, 1]
[1, 3, 4, 8]
[8, 4, 3, 1]
```

In [138…
```
# # Los elementos deben ser comparables para poderse ordenar
lista = [1, 'a']
lista.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[138], line 3
      1 # # Los elementos deben ser comparables para poderse ordenar
      2 lista = [1, 'a']
----> 3 lista.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

In [139…
```
compra = ['Huevos','Pan','zapallo','Leche','Licor']
print(sorted(compra))
print(compra)
```

```
['Huevos', 'Leche', 'Licor', 'Pan', 'zapallo']
['Huevos', 'Pan', 'zapallo', 'Leche', 'Licor']
```

- Pertenencia.

In [140…
```
lista = [1, 2, 3, 4]
print(1 in lista)
print(5 in lista)
```

```
True
False
```

- Anidamiento.

In [141…
```
letras = ['a', 'b', 'c', ['x', 'y', ['i', 'j', 'k']]]
print(letras[0])
print(letras[3][0])
print(letras[3][2][0])
```

```
a
x
i
```

In [142…
```
print(letras[3])
```

```
['x', 'y', ['i', 'j', 'k']]
```

In [143…
```
a =[1000,2,3]
b = a[:] #a.copy() #[:]
```

```
print(id(a))
print(id(b))

print(id(a[0]))
print(id(b[0]))
```

```
1631155680192
1631156359808
1631157469744
1631157469744
```

Alias/Referencias en las listas

- Las listas se pueden modificar, por lo que un cambio afecta al mismo objeto.
- Al pasar una lista a una función, los cambios dentro de la función también pueden afectar a la lista original.
- Para evitar problemas, se pueden hacer copias de las listas.

In [144…

```
lista = [2, 4, 16, 32]
ref = lista

print(id(lista))
print(id(ref))

ref[2] = 64

print(ref)
print(lista)
```

```
1631156068608
1631156068608
[2, 4, 64, 32]
[2, 4, 64, 32]
```

In [145…

```
lista = [2000, 4, 16, 32]
copia = lista[:]
copia = lista.copy()

copia[2] = 64

print(lista)
print(copia)

print(id(lista))
print(id(copia))

print(id(lista[2]))
print(id(copia[2]))
```

```
[2000, 4, 16, 32]
[2000, 4, 64, 32]
1631155681856
1631155971392
140711968494856
140711968496392
```

In [146…
```python
# #listas anidadas se copian por referencia
lista = [2, 4, 16, 32, [34, 10, [5,5]]]
copia = lista.copy()

copia[3] = 20
copia[4][0] = 28

print(copia)
print(lista)
```

```
[2, 4, 16, 20, [28, 10, [5, 5]]]
[2, 4, 16, 32, [28, 10, [5, 5]]]
```

In [147…
```python
# lista = [0, 1, [10, 20]]
# print(id(lista))
# lista2 = [10, 20]
# lista = [0, 1, lista2]
# copia = lista[:] #.copy()
# print(copia)
# print(id(copia))
# print(id(lista[2]))
# print(id(copia[2]))
# copia[2][0] = 40
# print(copia)
# print(lista)
```

In [148…
```python
# # evitar que listas anidadas se copien por referencia
# import copy
# lista = [2, 4, 16, 32, [34, 10, [5,5]]]
# copia = copy.deepcopy(lista)
# copia[0] = 454
# copia[4][2][0] = 64
# print(lista)
# print(copia)
# print(f"{id(lista)} - {id(copia)}")
# print(f"{id(lista[4])} - {id(copia[4])}")
```

3.2 Diccionarios

- Son colecciones de datos organizados en pares de clave y valor.
- Se pueden modificar después de crearlos.
- Claves:
  - Deben ser datos que no cambien.
  - No se repiten dentro del mismo diccionario.
- Valores:
- Pueden ser cualquier tipo de dato, como números, textos o listas.
- Desde Python 3.7, los elementos mantienen el orden en el que se agregan.

- Funcionan de forma parecida a una lista, pero en lugar de usar números como índice, usan claves.
- A diferencia de las listas, no se recorren como secuencias normales, sino que funcionan como asociaciones entre claves y valores.

Operaciones con diccionarios

- Creación de diccionarios.

In [149…
```
# Creación simple, usando una expresión literal.
persona = {'DNI' : '11111111D', 'Nombre' : 'Carlos', 'Edad' : 34}
print(persona)
```

```
{'DNI': '11111111D', 'Nombre': 'Carlos', 'Edad': 34}
```

In [150…
```
# Creación uniendo dos colecciones.
nombres = ['Pablo', 'Manolo', 'Pepe', 'Juan']
edades = [52, 14, 65]

datos = dict(zip(nombres,edades))
print(datos)
```

```
{'Pablo': 52, 'Manolo': 14, 'Pepe': 65}
```

In [151…
```
# Creación pasando claves y valores a la función 'dict'
persona2 = dict(nombre='Rosa', apellido='Garcia')
print(persona2)
```

```
{'nombre': 'Rosa', 'apellido': 'Garcia'}
```

In [152…
```
# Creación usando una lista de tuplas de dos elementos.

persona2 = dict([('nombre', 'Rosa'), ('apellido','Garcia')])
print(persona2)
```

```
{'nombre': 'Rosa', 'apellido': 'Garcia'}
```

In [153…
```
# Creación incremental por medio de asignación (como las claves no existen, se␣↪cr

persona = {}
print(persona['DNI'])
persona['DNI'] = '11111111D'
persona['Nombre'] = 'Carlos'
persona['Edad'] = 34
persona['DNI'] = '222222222'

print(persona)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[153], line 4
      1  # Creación incremental por medio de asignación (como las claves no existen,
se␣↪crean nuevos items)
      3 persona = {}
----> 4 print(persona[     ])
      5 persona['DNI'] = '11111111D'
      6 persona['Nombre'] = 'Carlos'

KeyError: 'DNI'
```

- Acceso a un valor a través de la clave.

In [154…
```
persona = {'DNI' : '11111111D', 'Nombre' : 'Carlos', 'Edad' : 34}
print(persona['Nombre'])
```

```
Carlos
```

In [155…
```
# Acceso a claves inexistentes o por índice produce error

# persona[1]
print(persona['Nombre'])

if 'Trabajo' in persona:
    print(persona['Trabajo'])
else:
    print("En el paro")

persona.get('Trabajo', "En el paro")
```

```
Carlos
En el paro
```

Out[155…
```
'En el paro'
```

- Modificación de un valor a través de la clave.

In [156…
```
persona = {'DNI' : '11111111D', 'Nombre' : 'Carlos', 'Edad' : 34}

persona['Nombre'] = 'Fernando'
persona['Edad'] += 1

print(persona)
```

```
{'DNI': '11111111D', 'Nombre': 'Fernando', 'Edad': 35}
```

- Añadir un valor a través de la clave.

In [157…
```
persona = {'DNI' : '11111111D', 'Nombre' : 'Carlos', 'Edad' : 34}
persona['Ciudad'] = 'Valencia'

print(persona)
```

```
{'DNI': '11111111D', 'Nombre': 'Carlos', 'Edad': 34, 'Ciudad': 'Valencia'}
```

- Eliminación de un valor a través de la clave.

In [158…
```
del persona['Ciudad']
value = persona.pop('Edad')

print(persona)
print(value)
```

```
{'DNI': '11111111D', 'Nombre': 'Carlos'}
34
```

- Comprobación de existencia de clave.

In [159…
```
print('Nombre' in persona)
print('Apellido' in persona)
```

```
True
False
```

- Recuperación del valor de una clave, indicando valor por defecto en caso de ausencia.

In [160…
```
persona = {'Nombre' : 'Carlos'}

value = persona.get('Nombre')
print(value)

# persona['Estatura']

value = persona.get('Estatura', 180)
print(value)
```

```
Carlos
180
```

- Anidamiento.

In [161…
```
persona = {
'Trabajos' : ['desarrollador', 'gestor'],
'Direccion' : {'Calle' : 'Pintor Sorolla', 'Ciudad' : 'Valencia'}
}

print(persona['Direccion'])
print(persona['Direccion']['Ciudad'])
```

```
{'Calle': 'Pintor Sorolla', 'Ciudad': 'Valencia'}
Valencia
```

- Métodos items, keys y values.

In [162…
```python
persona = {'DNI' : '11111111D', 'Nombre' : 'Carlos', 'Edad' : 34}

print(list(persona.items()))
print(list(persona.keys()))
print(list(persona.values()))
```

```
[('DNI', '11111111D'), ('Nombre', 'Carlos'), ('Edad', 34)]
['DNI', 'Nombre', 'Edad']
['11111111D', 'Carlos', 34]
```

In [163…
```python
dict_simple = {'ID' : 'XCSDe1194', 'Nombre' : 'Carlos', 'Edad' : 34}

# iterar sobre las claves
print('Claves\n')
for key in dict_simple.keys(): # o dict_simple
    print(key)

print('\nValores\n')
# iterar sobre los valores
for value in dict_simple.values():
    print(value)
```

```
Claves

ID
Nombre
Edad

Valores

XCSDe1194
Carlos
34
```

In [164…
```python
dict_simple = {'ID' : 'XCSDe1194', 'Nombre' : 'Carlos', 'Edad' : 34}
# iterar sobre ambos, directamente con items().
for key, value in dict_simple.items():
    print('Clave: ' + str(key) + ', valor: ' + str(value))

for item in dict_simple.items(): # devuelve tuplas
    print('Item: ' + str(item))

# usar zip para iterar sobre Clave-Valor
for key, value in zip(dict_simple.keys(),dict_simple.values()):
    print('Clave: ' + str(key) + ', valor: ' + str(value))
```

```
Clave: ID, valor: XCSDe1194
Clave: Nombre, valor: Carlos
Clave: Edad, valor: 34
Item: ('ID', 'XCSDe1194')
Item: ('Nombre', 'Carlos')
Item: ('Edad', 34)
Clave: ID, valor: XCSDe1194
Clave: Nombre, valor: Carlos
Clave: Edad, valor: 34
```

3.3 Sets (Conjuntos)

Al igual que las listas:

- Son colecciones de elementos.
- Pueden contener diferentes tipos de datos.
- Se pueden modificar.
- No tienen un tamaño fijo y pueden crecer según la memoria disponible.

A diferencia de las listas:

- No permiten elementos repetidos.
- Se escriben usando llaves { }.
- Sus elementos no tienen un orden definido.
- Solo pueden guardar elementos que no se puedan modificar.
- No pueden contener otros conjuntos o listas dentro.

Operaciones con conjuntos

- Creación de conjuntos.

In [165…
```
set1 = {0, 1, 1, 2, 3, 4, 4}
print(set1)

set2 = {'user1', 12, 2}
print(set2)

set3 = set(range(7))
print(set3)

set4 = set([0, 1, 2, 3, 4, 0, 1])
print(set4)
```

```
{0, 1, 2, 3, 4}
{2, 'user1', 12}
{0, 1, 2, 3, 4, 5, 6}
{0, 1, 2, 3, 4}
```

In [166…
```
 #Observa la diferencia entre listas y conjuntos
s = 'aabbc'
print(list(s))
print(set(s))
```

```
['a', 'a', 'b', 'b', 'c']
{'a', 'c', 'b'}
```

- Acceso por índice genera error.

In [167…
```
set1 = {0, 1, 2}
print(set1[0])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[167], line 2
      1 set1 = {0, 1, 2}
----> 2 print(set1[0])

TypeError: 'set' object is not subscriptable
```

- Unión, intersección y diferencia.

In [168…
```
set1 = {0, 1, 1, 2, 3, 4, 5, 8, 13, 21}
set2 = set([0, 1, 2, 3, 4, 42])

# union
print(set1 | set2)

# intersección
print(set1 & set2)

# diferencia
print(set1 - set2)
print(set2 - set1)
```

```
{0, 1, 2, 3, 4, 5, 8, 42, 13, 21}
{0, 1, 2, 3, 4}
{8, 13, 21, 5}
{42}
```

In [169…
```
a = [1,2]
b = [2,3]
print(set(a) & set(b))
```

```
{2}
```

In [170…
```
#Además de los operadores, que operan únicamente con Sets, también se pueden␣↪usar
conjunto = {0, 1, 2}
lista = [1, 3, 3]

print(conjunto.union(lista))
print(conjunto.intersection(lista))
print(conjunto.difference(lista))
```

```
{0, 1, 2, 3}
{1}
{0, 2}
```

- comparación de conjuntos.

In [171…
```
set1 = {0, 1, 1, 2, 3, 4, 5, 8, 13, 21}
set2 = set([0, 1, 2, 3, 4])

print(set2.issubset(set1))
print(set1.issuperset(set2))
print(set1.isdisjoint(set2))
```

```
True
True
False
```

- Pertenencia.

In [172…
```
words = {'calm', 'balm'}
print('calm' in words)
```

```
True
```

- Anidamiento.

In [173…
```
# Los conjuntos no soportan anidamiento, pero como permite elementos␣↪inmutables, s

nested_set = {1, (1, 1, 1), 2, 3, (1,1,1)}
print(nested_set)
```

```
{1, 2, 3, (1, 1, 1)}
```

- Modificación de conjuntos.

In [174…
```
# A través de operador de asignación

set1 = {'a', 'b', 'c'}
set2 = {'a', 'd'}

# set1 |= set2 # set1 = set1 | set2
# set1 &= set2
set1 -= set2

print(set1)
```

```
{'c', 'b'}
```

In [175…
```
# A través de método 'update'.

set1 = {'a', 'b', 'c'}
set2 = {'a', 'd'}

# set1.update(set2)
# set1.intersection_update(set2)
set1.difference_update(set2)

print(set1)
```

```
{'c', 'b'}
```

In [176…
```
# A través de métodos 'add' y 'remove'.

set1 = {'a', 'b', 'c'}

set1.add('d')
```

```
set1.remove('a')

print(set1)
```

```
{'c', 'b', 'd'}
```

3.4 Tuplas

Al igual que las listas:

- Son conjuntos de elementos.
- Pueden guardar distintos tipos de datos.
- Admiten elementos repetidos.
- Pueden tener muchos elementos, según la memoria disponible.
- Los elementos están ordenados y cada uno tiene una posición.
- Se accede a los elementos usando un índice.
- Los índices empiezan en 0 y llegan hasta el último elemento.
- El orden de los elementos es importante.
- Pueden contener otras colecciones dentro (anidamiento).

A diferencia de las listas:

- Se escriben usando paréntesis.
- No se pueden modificar una vez creadas.

¿Para qué sirven las tuplas?

Para guardar grupos de datos que no deben cambiar, como una fecha.

Son útiles cuando se necesita que los datos no se modifiquen, por ejemplo como claves en un diccionario.

Operaciones con tuplas

- Creación de tuplas.

In [177…

```
tuple1 = ('Foo', 34, 5.0, 34)
print(tuple1)

tuple2 = 1, 2, 3
print(tuple2)

tuple3 = tuple(range(10))
print(tuple3)

tuple4 = tuple([0, 1, 2, 3, 4])
print(tuple4)
```

```
('Foo', 34, 5.0, 34)
(1, 2, 3)
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
(0, 1, 2, 3, 4)
```

In [178…
```
# Ojo con las tuplas de un elemento. Los paréntesis se intepretan como␣↪indicadores
singleton_number = (1)
type(singleton_number)
```

Out[178…
```
int
```

In [179…
```
# Creación de tupla de un elemento.
singleton_tuple = (1,)
print(type(singleton_tuple))
print(singleton_tuple)
```

```
<class 'tuple'>
(1,)
```

- Obtención del número de elementos.

In [180…
```
print(len(tuple1))
```

```
4
```

- Acceso por índice

In [181…
```
tuple1 = ('Foo', 1, 2, 3)

print(tuple1[0]) # Primer elemento
print(tuple1[len(tuple1)-1]) # Último elemento

print(tuple1[-1]) # Índices negativos comienzan desde el final
print(tuple1[-len(tuple1)]) # primer elemento
```

```
Foo
3
3
Foo
```

- Asignación a tuplas falla. Son immutables

In [182…
```
# tuple1[0] = 'bar'
```

In [183…
```
# print(id(tuple1))
# lista = list(tuple1)
# lista[0] = 'bar'
# tuple1 = tuple(lista)
# print(id(tuple1))

# print(tuple1)
```

- Contar número de ocurrencias de un elemento.

In [184…
```
 # tuple1 = ('Foo',34, 5.0, 34)
# print(tuple1.count(34))
```

- Encontrar el índice de un elemento.

In [185…
```
# tuple1 = ('Foo', 34, 5.0, 34)
# indice = tuple1.index(34)

# print(indice)
# print(tuple1[indice])
```

In [186…
```
# Si el elemento no existe, error

# tuple1 = ('Foo', 34, 5.0, 34)
# print(tuple1.index(1))
```

In [187…
```
# comprobar si existe antes
# tuple1 = ('Foo',34, 5.0, 34)

# elemento = 35
#    if elemento in tuple1:
# print(tuple1.index(elemento))
# else:
#    print(str(elemento) + ' not found')
```

- Desempaquetar una tupla.

In [188…
```
# tuple1 = (1, 2, 3, 4)
# a, b, c, d = tuple1 # a,b,c - a,b,c,d - a,b,_,_ - a, *_

# print(a)
# print(b)
# print(_)
# print(d)
```

In [189…
```
# a, b, *resto = tuple1
# print(a)
# print(b)
# print(resto)
```

3.5 Secuencias

- Son objetos que se pueden recorrer elemento por elemento.
- No son listas, pero se pueden convertir en listas usando list().
- Algunos ejemplos son range y enumerate.

In [190…
```
# list(range(0,10,2))
```

In [191…
```
# lista = [10, 20, 30]
# print(list(enumerate(lista)))
```

In [192…
```
# enumerate para iterar una colección (índice y valor)
# lista = [10, 20, 30]
# for index, value in enumerate(lista):
#    print('Index = ' + str(index) + '. Value = ' + str(value))

# for index, value in [(0,10), (1,20), (2,30)]:
#    print('Index = ' + str(index) + '. Value = ' + str(value))
```

In [193…
```
# enumerate para iterar una colección (índice y valor)
# for index, value in enumerate(range(0,10,2)):
#    print('Index = ' + str(index) + '. Value = ' + str(value))

# for value in enumerate(range(0,10,2)):
#    print(value)
```

In [194…
```
 # zip para unir, elemento a elemento, dos colecciones, retornando lista de␣↪tuplas
# útil para iterar dos listas al mismo tiempo
# nombres = ['Manolo','Pepe','Luis']
# edades = [31,34,34,45]

# for nombre,edad in zip(nombres,edades):
#    print('Nombre: ' + nombre + ', edad: ' + str(edad))
```

In [195…
```
# nombres = ['Manolo','Pepe','Luis']
# edades = [31,34,34]

# for i in range(0,len(nombres)):
#    print(f"Nombre es {nombres[i]} y edad es {edades[i]}")
```

In [196…
```
# nombres = ['Manolo','Pepe','Luis']
# edades = [31,34,34]
# jugadores = zip(nombres,edades) # genera secuencia
# print(list(jugadores))
# print(type(jugadores))
```

In [197…
```
# listas de diferentes longitudes
# nombres = ['Manolo','Pepe','Luis']
# edades = [31,34,34,44,33]
# jugadores = zip(nombres,edades)
# print(list(jugadores))
```

In [198…
```
# unzip
# nombres = ['Manolo','Pepe','Luis']
# edades = [31,34,34]
# alturas = [120, 130, 140]
# jugadores = zip(nombres,edades, alturas)
# # print(list(jugadores))
```

```
# ns, es, al = zip(*jugadores)
# print(list(ns))
# print(es)
# print(al)
```

In [199…

```
# nombres = ['Manolo','Pepe','Luis']
# edades = [31,34,34]
# dd = [23,34,45]
# jugadores_z = zip(nombres,edades, dd)
# # print(list(jugadores_z))

# jugadores_uz = zip(*jugadores_z)
# print(list(jugadores_uz))
```

3.6 En Python todo son objetos

- Identidad: Es una forma de identificar al objeto desde que se crea y no cambia. Sirve para saber si dos objetos son exactamente el mismo.
- Tipo: Indica qué clase de objeto es y qué cosas se pueden hacer con él. Este no cambia.
- Valor: Es la información que guarda el objeto. Algunos valores se pueden modificar y otros no.

– Mutables: listas, diccionarios, conjuntos y objetos creados por el usuario. – Inmutables: números, valores lógicos, textos y tuplas.

In [200…

```
# Asignación

# list_numbers = [1, 2, 3] # Lista (mutable)
# tuple_numbers = (1, 2, 3) # Tupla (inmutable)

# print(list_numbers[0])
# print(tuple_numbers[0])

# list_numbers[0] = 100
# # tuple_numbers[0] = 100

# print(list_numbers)
# print(tuple_numbers)

# tuple_l_numbers = list(tuple_numbers)
# tuple_l_numbers[0] = 100
# tuple_numbers = tuple(tuple_l_numbers)
# print(tuple_numbers)
```

In [201…

```
# Identidad

# list_numbers = [1, 2, 3]
# tuple_numbers = (1, 2, 3)

# print('Id list_numbers: ' + str(id(list_numbers)))
# print('Id tuple_numbers: ' + str(id(tuple_numbers)))
```

```
# list_numbers += [4, 5, 6] # La lista original se extiende
# tuple_numbers += (4, 5, 6) # Se crea un nuevo objeto

# print(list_numbers)
# print(tuple_numbers)

# print('Id list_numbers: ' + str(id(list_numbers)))
# print('Id tuple_numbers: ' + str(id(tuple_numbers)))
```

In [202…

```
# Referencias

# list_numbers = [1, 2, 3]
# list_numbers_2 = list_numbers # list_numbers_2 referencia a list_numbers

# print('Id list_numbers: ' + str(id(list_numbers)))
# print('Id list_numbers_2: ' + str(id(list_numbers_2)))

# list_numbers.append(4) # Se actualiza list_numbers2 también

# print(list_numbers)
# print(list_numbers_2)

# print('Id list_numbers: ' + str(id(list_numbers)))
# print('Id list_numbers_2: ' + str(id(list_numbers_2)))
```

In [203…

```
# text = "Hola" # Inmutable
# text_2 = text # Referencia

# print('Id text: ' + str(id(text)))
# print('Id text_2: ' + str(id(text_2)))

# text += " Mundo"

# print(text)
# print(text_2)

# print('Id text: ' + str(id(text)))
# print('Id text_2: ' + str(id(text_2)))
```

In [204…

```
# teams = ["Team A", "Team B", "Team C"] # Mutable
# player = (23, teams) # Inmutable

# print(type(player))
# print(player)
# print(id(player))

# teams[2] = "Team J"

# print(player)
# print(id(player))

# player[1][0] = 'Team XX'
# print(teams)
```

3.7 Ejercicios

1. Escribe un programa que muestre por pantalla la concatenación de un número y una cadena de caracteres. Para obtener esta concatenación puedes usar uno de los operadores explicados en este tema. Ejemplo: dado el número 3 y la cadena 'abc', el programa mostrará la cadena '3abc'.
2. Escribe un programa que muestre por pantalla un valor booleano que indique si un número entero N está contenido en un intervalo semiabierto [a,b), el cual establece una cota inferior a (inclusive) y una cota superior b (exclusive) para N.
3. Escribe un programa que, dado dos strings S1 y S2 y dos números enteros N1 y N2, determine si el substring que en S1 se extiende desde la posición N1 a la N2 (ambos inclusive) está contenido en S2.
4. Dada una lista con elementos duplicados, escribir un programa que muestre una nueva lista con el mismo contenido que la primera pero sin elementos duplicados.
5. Escribe un programa que, dada una lista de strings L, un string s perteneciente a L y un string t, reemplace s por t en L. El programa debe mostrar la lista resultante por pantalla.
6. Escribe un programa que defina una tupla con elementos numéricos, reemplace el valor del último por un valor diferente y muestre la tupla por pantalla. Recuerda que las tuplas son inmutables. Tendrás que usar objetos intermedios.
7. Dada la lista [1,2,3,4,5,6,7,8] escribe un programa que, a partir de esta lista, obtenga la lista [8,6,4,2] y la muestre por pantalla.
8. Escribe un programa que, dada una tupla y un índice válido i, elimine el elemento de la tupla que se encuentra en la posición i. Para este ejercicio sólo puedes usar objetos de tipo tupla. No puedes convetir la tupla a una lista, por ejemplo.
9. Escribe un programa que obtenga la mediana de una lista de números. Recuerda que la mediana M de una lista de números L es el número que cumple la siguiente propiedad: la mitad de los números de L son superiores a M y la otra mitad son inferiores. Cuando el número de elementos de L es par, se puede considerar que hay dos medianas. No obstante, en este ejercicio consideraremos que únicamente existe una mediana.

4. Soluciones

In [223…

```python
# Definimos el número
numero = 3

# Definimos la cadena de texto
texto = "abc"

# Convertimos el número a texto y lo concatenamos
resultado = str(numero) + texto

# Mostramos el resultado por pantalla
print(resultado)
```

```
3abc
```

In [224…
```python
# Número a evaluar
N = 5

# Límite inferior (incluido)
a = 3

# Límite superior (no incluido)
b = 8

# Comprobamos si N está dentro del intervalo
print(a <= N < b)
```

```
True
```

In [225…
```python
# Strings dados
S1 = "programacion"
S2 = "estoy estudiando programacion en python"

# Posiciones inicial y final
N1 = 0
N2 = 6

# Extraemos el substring desde N1 hasta N2 (inclusive)
substring = S1[N1:N2 + 1]

# Verificamos si está contenido en S2
print(substring in S2)
```

```
True
```

In [226…
```python
# Lista con elementos duplicados
lista = [1, 2, 3, 2, 4, 1, 5]

# Lista vacía para guardar valores únicos
sin_duplicados = []

# Recorremos la lista original
for elemento in lista:
    if elemento not in sin_duplicados:
        sin_duplicados.append(elemento)

# Mostramos la nueva lista
print(sin_duplicados)
```

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

In [227…
```python
# Lista de strings
L = ["rojo", "azul", "verde"]

# String a reemplazar
s = "azul"

# Nuevo string
t = "amarillo"
```

```
# Recorremos la lista
for i in range(len(L)):
    if L[i] == s:
        L[i] = t

# Mostramos la lista resultante
print(L)
```

```
['rojo', 'amarillo', 'verde']
```

In [228…
```
# Tupla original
numeros = (10, 20, 30)

# Creamos una nueva tupla cambiando el último valor
nueva_tupla = numeros[0:2] + (99,)

# Mostramos la nueva tupla
print(nueva_tupla)
```

```
(10, 20, 99)
```

In [229…
```
# Lista original
lista = [1, 2, 3, 4, 5, 6, 7, 8]

# Tomamos los valores desde el final, de dos en dos
resultado = lista[::-2]

# Mostramos la nueva lista
print(resultado)
```

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

In [230…
```
# Tupla original
tupla = (10, 20, 30, 40)

# Índice del elemento a eliminar
i = 2

# Creamos una nueva tupla sin ese elemento
nueva_tupla = tupla[:i] + tupla[i + 1:]

# Mostramos la nueva tupla
print(nueva_tupla)
```

```
(10, 20, 40)
```

In [231…
```
# Lista de números
numeros = [5, 1, 3]

# Ordenamos la lista
numeros.sort()

# Obtenemos el valor central (mediana)
mediana = numeros[len(numeros) // 2]

# Mostramos la mediana
print(mediana)
```

5. Github

https://github.com/Erick-305/machine-learning.git

In [ ]: