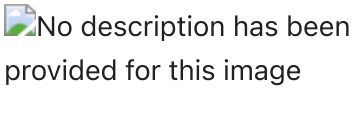

Data Types

Nombre: Edgar Garces
Fecha: 08/01/2026

1 Comentarios

1.0.1 ¿Qué son?

Texto contenido en ficheros Python que es ignorado por el intérprete; es decir, no es ejecutado.

1.0.2 ¿Cuál es su utilidad?

• Se puede utilizar para documentar código y hacerlo más legible
• Preferiblemente, trataremos de hacer código fácil de entender y que necesite pocos comentar ios, en lugar de vernos forzados a recurrir a los comentarios para explicar el código.

1.0.3 Tipos de comentarios

Comentarios de una línea
• Texto precedido por '#'
• Se suele usar para documentar expresiones sencillas.

In [1]:
```
# 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 [2]:
```
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 [3]:
```
4+3
```

Out[3]:
```
7
```

2.1 Literales

• Python tiene una serie de tipos de datos integrados en el propio lenguaje.
• Los literales son expresiones que generan objetos de estos tipos.
• Estos objetos, según su tipo, pueden ser:
– Simples o compuestos.
– Mutables o immutables.

Literales simples

• Enteros
• Decimales o punto flotante
• Booleano

In [4]:
```
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 [5]:
```
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[5]: (2, 4)

2.2 Variables

- Referencias a objetos.
- Las variables y los objetos se almacenan en diferentes zonas de memoria.
- Las variables siempre referencian a objetos y nunca a otras variables.
- Objetos sí que pueden referenciar a otros objetos. Ejemplo: listas.
- Sentencia de asignación:

<nombre_variable> '='

In [6]:
```
# # Asignación de variables
a = 5
print(a)
```

```
5
```

In [7]:
```
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 [8]:
```
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.*

Identificadores

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

In [9]:
```
a = 3
print(id(a))
a = 'Pablo García'
print(id(a))
a = 4.5
print(id(a))
```

```
4368875200
4564443536
4563701424
```

• Referencias compartidas: un mismo objeto puede ser referenciado por más de una variable.

– Variables que referencian al mismo objeto tienen mismo identificador.

In [10]:
```
a = 4567
print(id(a))
```

```
4563707984
```

In [11]:
```
b = a
print(id(b))
```

```
4563707984
```

In [12]:
```
c = 4567
print(id(c))
```

```
4563708720
```

In [13]:
```
a = 25
b = 25
print(id(a))
print(id(b))
print(id(25))
```

```
4368875904
4368875904
4368875904
```

In [14]:
```
# # Ojo con los enteros "grandes" [-5, 256]
a = 258
b = 258
print(id(a))
print(id(b))
print(id(258))
```

```
4563707600
4563708688
4563708976
```

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

In [15]:
```
a = 400
b = a
print(id(a))
print(id(b))
```

```
4563708752
4563708752
```

• Las variables pueden aparecer en expresiones.

In [16]:
```
a = 3
b = 5
```

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

```
4368875200
8
```

In [17]:
```
c = a + b
print(c)
print(id(c))
```

```
8
4368875360
```

Respecto a los nombres de las variables ...

- No se puede poner números delante del nombre de las variables.
- Por convención, evitar CamelCase. Mejor usar snake_case: uso de "_" para separar palabras.
- El lenguaje diferencia entre mayúsculas y minúsculas.
- Deben ser descriptivos.
- Hay palabras o métodos reservados -> Built-ins y KeyWords

– Ojo con reasignar un nombre reservado!

In [18]:
```
print(pow(3,2))
```

```
9
```

In [19]:
```
#print(pow(3,2))

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

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

In [20]:
```
def pow(a, b):
    return a + b
print(pow(3,2))
```

```
5
```

Asignación múltiple de variables

In [21]:
```
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 [22]:
```
a = 1
b = 2
a, b = b, a
print(a, b)
```

```
2 1
```

In [23]:
```
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 [24]:
```
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 [25]:
```
a = True
b = False
print(id(a))
print(id(b))
print(a is b)
print(a == b)
```

```
4368874064
4368874032
False
False
```

Números

In [26]:
```
print(2) # Enteros, sin parte fraccional.
print(3.4) # Números en coma flotante, con parte fraccional.
print(2+4j) # Números complejos.
print(1/2) # Numeros racionales.
```

```
2
3.4
(2+4j)
0.5
```

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

In [27]:
```python
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 [28]:
```python
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 c
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 [29]:
```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 [30]:
```
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'
```

```
1.75
1
1
```

2.5 Operadores

Python3 precedencia en operaciones

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

Operatores aritméticos

Operador	Descripción
a + b	Suma
a - b	Resta
a / b	División
a // b	División entera
a % b	Módulo / Resto
a * b	Multiplicación
a ** b	Exponenciación

In [31]:
```
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	Descripción
a > b	Mayor
a < b	Menor
a == b	Igualdad
a != b	Desigualdad
a >= b	Mayor o igual
a <= b	Menor o igual

In [32]:
```
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	Descripción
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

Enlace a Tablas de Verdad.

In [33]:
```
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 | Descripción | |----------|---| | a & b | AND binario | | a | b | OR binario | | a ^ b | XOR binario | | ~a | NOT binario | | a >> b | Desplazamiento binario a la derecha | | a << b | Desplazamiento binario a la izquierda |

In [34]:
```
# 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 | Descripción | |----------|---| | = | Asignación | | += | Suma y asignación | | -= | Resta y asignación | | *= | Multiplicación y asignación | | /= | División y asignación | | %= | Módulo y asignación | | //= | División entera y asignación | | **= | Exponenciación y asignación | | &= | AND binario y asignación | | |= | OR binario y asignación | | ^= | XOR binario y asignación | | >>= | Desplazamiento a la derecha y asignación | | <<= | Desplazamiento a la izquierda y asignación |

In [35]:
```
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	Descripción
`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 [36]:
```
a = 4444
b = a
print(a is b)
print(a is not b)
```

```
True
False
```

Operadores de Pertenencia

Operador	Descripción
`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 [37]:
```
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 [38]:
```
valor = input("Inserte valor:")
print(valor)
print(type(valor))
```

```
5
<class 'str'>
```

In [39]:
```
grados_c = int(input("Conversión de grados a fahrenheit, inserte un valor: "
print(f"Grados F: {1.8 * (grados_c) + 32}")
```

```
Grados F: 41.0
```

3 Tipos de datos compuestos (colecciones)

3.1 Listas

- Una colección de objetos.
- Mutables.
- Tipos arbitrarios heterogeneos.
- Puede contener duplicados.

11

- 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 [40]:
```
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 [41]:
```python
# # 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 [42]:
```python
# # 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 0x11041a2a0>]
```

In [43]:
```python
# # 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 [44]:
```python
letras = ['a', 'b', 'c', 'd', 1]
print(len(letras))
```

```
5
```

- Acceso a un elemento de una lista

In [45]:
```python
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 [46]:
```python
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 [47]:
```
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']
4568080704
['a', 'b', 'c', 'd']
4568080640
['a', 'b', 'c', 'd']
4568079168
```

- Añadir un elemento al final de la lista

In [48]:
```
letras.append('e')
print(letras)
print(id(letras))
```

```
['a', 'b', 'c', 'd', 'e']
4568080704
```

In [49]:
```
letras += 'e'
print(letras)
print(id(letras))
```

```
['a', 'b', 'c', 'd', 'e', 'e']
4568080704
```

- Insertar en posición.

In [50]:
```
print(len(letras))
letras.insert(1,'g')
print(len(letras))
print(letras)
print(id(letras))
```

```
6
7
['a', 'g', 'b', 'c', 'd', 'e', 'e']
4568080704
```

- Modificación de la lista (individual).

In [51]:
```
letras[5] = 'f'
print(letras)
```

```
print(id(letras))
```

```
['a', 'g', 'b', 'c', 'd', 'f', 'e']
4568080704
```

In [52]:
```
# # index tiene que estar en rango
#letras[20] = 'r'
```

- Modificación múltiple usando slicing.

In [53]:
```
letras = ['a', 'b', 'c', 'f', 'g', 'h', 'i', 'j', 'k']
print(id(letras))
```

```
4567713088
```

In [54]:
```
letras[0:7:2] = ['z', 'x', 'y','p']
print(letras)
# print(id(letras))
```

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

In [55]:
```
# # Ojo con la diferencia entre modificación individual y múltiple. Asignaci
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 [56]:
```
#letras.remove('f')
if 'p' in letras:
    letras.remove('p')
# #letras.remove('z')
print(letras)
```

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

In [57]:
```
# 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 [58]:
```
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 [59]:
```
# numeros1=10
# print(numeros1)
# print(numeros)
```

In [60]:
```
#lista = []
#a = lista.pop()
```

- Encontrar índice de un elemento.

In [61]:
```
letras = ['a', 'b', 'c', 'c']
if 'a' in letras:
    print(letras.index('a'))
    print(letras)
```

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

- Concatenar listas.

In [62]:
```
lacteos = ['queso', 'leche']
frutas = ['naranja', 'manzana']
print(id(lacteos))
print(id(frutas))
compra = lacteos + frutas
print(id(compra))
print(compra)
```

```
4568080832
4568081088
4568157184
['queso', 'leche', 'naranja', 'manzana']
```

In [63]:
```
# # Concatenación sin crear una nueva lista
frutas = ['naranja', 'manzana']
print(id(frutas))
frutas.extend(['pera', 'uvas'])
print(frutas)
print(id(frutas))
```

```
4568074496
['naranja', 'manzana', 'pera', 'uvas']
4568074496
```

In [64]:
```
# # Anidar sin crear una nueva lista
frutas = ['naranja', 'manzana']
print(id(frutas))
frutas.append(['pera', 'uvas'])
```

```
print(frutas)
print(id(frutas))
```

```
4567924928
['naranja', 'manzana', ['pera', 'uvas']]
4567924928
```

- Replicar una lista.

In [65]:
```
lacteos = ['queso', 'leche']
print(lacteos * 3)
print(id(lacteos))
a = 3 * lacteos
print(a)
print(id(a))
```

```
['queso', 'leche', 'queso', 'leche', 'queso', 'leche']
4568157376
['queso', 'leche', 'queso', 'leche', 'queso', 'leche']
4568158208
```

- Copiar una lista

In [66]:
```
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 = 4567924928
id frutas2 = 4568074496
```

- Ordenar una lista.

In [67]:
```
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 [68]:
```
# # Los elementos deben ser comparables para poderse ordenar
lista = [1, 'a']
lista.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[68], 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 [ ]:
```
compra = ['Huevos','Pan','zapallo','Leche','Licor']
print(sorted(compra))
print(compra)
```

- Pertenencia.

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

- Anidamiento.

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

In [ ]:
```
print(letras[3])
```

In [ ]:
```
a =[1000,2,3]
b = a[:] #a.copy() #[:]
print(id(a))
print(id(b))
print(id(a[0]))
print(id(b[0]))
```

Alias/Referencias en las listas

- Son mutables y las asignaciones a un objeto alteran el primero
- Pasar listas a funciones puede suponer un riesgo
- Clonar o copiar listas

In [ ]:
```
lista = [2, 4, 16, 32]
ref = lista
print(id(lista))
print(id(ref))
ref[2] = 64
print(ref)
print(lista)
```

In [ ]:
```
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]))
```

In [ ]:
```
# #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)
```

In [ ]:
```
# 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 [ ]:
```
# # 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

- Colección de parejas clave-valor.
- Son mutables.
- Claves:

– Cualquier objeto inmutable.

– Sólo pueden aparecer una vez en el diccionario.

- Valores:

– Sin restricciones. Cualquier objeto (enteros, strings, listas, etc.) puede hacer de valor.

- Desde la versión 3.7 están ordenados.
- Se pueden ver como listas indexadas por cualquier objeto inmutable, no

necesariamente por números enteros.

- A diferencia de las listas, no son secuencias. Son mappings.

Operaciones con diccionarios

- Creación de diccionarios.

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

In [ ]:
```
# Creación uniendo dos colecciones.
nombres = ['Pablo', 'Manolo', 'Pepe', 'Juan']
edades = [52, 14, 65]
datos = dict(zip(nombres,edades))
print(datos)
```

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

In [ ]:
```
# Creación usando una lista de tuplas de dos elementos.
persona2 = dict([('nombre', 'Rosa'), ('apellido','Garcia')])
print(persona2)
```

In [ ]:
```
# Creación incremental por medio de asignación (como las claves no existen,
↪crean nuevos items)
persona = {}
print(persona['DNI'])
persona['DNI'] = '11111111D'
persona['Nombre'] = 'Carlos'
persona['Edad'] = 34
persona['DNI'] = '222222222'
print(persona)
```

- Acceso a un valor a través de la clave.

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

In [ ]:
```
# 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")
```

- Modificación de un valor a través de la clave.

In [ ]:
```
persona = {'DNI' : '11111111D', 'Nombre' : 'Carlos', 'Edad' : 34}
persona['Nombre'] = 'Fernando'
persona['Edad'] += 1
print(persona)
```

- Añadir un valor a través de la clave.

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

- Eliminación de un valor a través de la clave.

In [ ]:
```
del persona['Ciudad']
value = persona.pop('Edad')
print(persona)
print(value)
```

- Comprobación de existencia de clave.

In [ ]:
```
print('Nombre' in persona)
print('Apellido' in persona)
```

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

In [ ]:
```
persona = {'Nombre' : 'Carlos'}
value = persona.get('Nombre')
print(value)
# persona['Estatura']
value = persona.get('Estatura', 180)
print(value)
```

- Anidamiento.

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

- Métodos items, keys y values.

In [ ]:
```
persona = {'DNI' : '11111111D', 'Nombre' : 'Carlos', 'Edad' : 34}
print(list(persona.items()))
print(list(persona.keys()))
print(list(persona.values()))
```

In [ ]:
```
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)
```

In [ ]:
```
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))
```

3.3 Sets (Conjuntos)

Al igual que las listas:

- Colección de elementos.
- Tipos arbitrarios.
- Mutables.
- No tienen tamaño fijo. Pueden contener tantos elementos como quepan en memoria.

A diferencia de las listas:

- No puede tener duplicados.
- Se definen por medio de llaves.
- Los elementos no van ordenados por posición. No hay orden establecido.
- Solo pueden contener objetos inmutables.
- No soportan anidamiento.

Operaciones con conjuntos

- Creación de conjuntos.

In [ ]:
```
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)
```

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

- Acceso por índice genera error.

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

- Unión, intersección y diferencia.

In [ ]:
```
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)
```

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

In [ ]:
```
#Además de los operadores, que operan únicamente con Sets, también se pueden
↪usar métodos que pueden operar sobre cualquier objeto iterable.
conjunto = {0, 1, 2}
lista = [1, 3, 3]
print(conjunto.union(lista))
print(conjunto.intersection(lista))
print(conjunto.difference(lista))
```

- comparación de conjuntos.

In [ ]:
```
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))
```

- Pertenencia.

In [ ]:
```
words = {'calm', 'balm'}
print('calm' in words)
```

- Anidamiento.

In [ ]:
```
# Los conjuntos no soportan anidamiento, pero como permite elementos␣
↪inmutables, se pueden "anidar" tuplas.
nested_set = {1, (1, 1, 1), 2, 3, (1,1,1)}
print(nested_set)
```

- Modificación de conjuntos.

In [ ]:
```
# A través de operador de asignación
set1 = {'a', 'b', 'c'}
set2 = {'a', 'd'}
```

```
# set1 |= set2 # set1 = set1 | set2
# set1 &= set2
set1 -= set2
print(set1)
```

In [ ]:
```
# 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)
```

In [ ]:
```
# A través de métodos 'add' y 'remove'.
set1 = {'a', 'b', 'c'}
set1.add('d')
set1.remove('a')
print(set1)
```

3.4 Tuplas

Al igual que las listas:

- Colección de elementos.
- Tipos arbitrarios.
- Puede contener duplicados.
- No tienen tamaño fijo. Pueden contener tantos elementos como quepan en memoria.
- Los elementos van ordenados por posición.
- Acceso a través de la sintaxis: [index]
- Índices van de 0 a n-1, donde n es el número de elementos de la tupla.
- Son secuencias donde el orden de los elementos importa.
- Soportan anidamiento.

A diferencia de las listas:

- Se definen por medio de paréntesis.
- Inmutables.

¿Por qué tuplas?

- Representación de una colección fija de elementos (por ejemplo, una fecha).
- Pueden usarse en contextos que requieren inmutabilidad (por ejemplo, como claves de un diccionario).

Operaciones con tuplas

- Creación de tuplas.

In [ ]:
```
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)
```

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

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

- Obtención del número de elementos.

In [ ]:
```
print(len(tuple1))
```

- Acceso por índice.

In [ ]:
```
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
```

- Asignación a tuplas falla. Son immutables.

In [ ]:
```
# tuple1[0] = 'bar'
```

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

- Encontrar el índice de un elemento.

In [ ]:
```
# tuple1 = ('Foo', 34, 5.0, 34)
# indice = tuple1.index(34)
# print(indice)
# print(tuple1[indice])
```

In [ ]:
```
# Si el elemento no existe, error
# tuple1 = ('Foo', 34, 5.0, 34)
```

```
# print(tuple1.index(1))
```

In [ ]:
```
# 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 [ ]:
```
# 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 [ ]:
```
# a, b, *resto = tuple1
# print(a)
# print(b)
# print(resto)
```

In [ ]:
```
3.5 Secuencias
• Formalmente, objetos iterables no materializados
• No son listas. 'list()' para materializarla
• 'range' o 'enumerate'
```

In [ ]:
```
# list(range(0,10,2))
```

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

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

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

In [ ]:
```
# 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 [ ]:
```
# 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 Para terminar, volvemos a enfatizar un matiz importante ...

En Python todo son objetos

Cada objeto tiene:

• Identidad: Nunca cambia una vez creado. Es como la dirección de memoria. Operador is compara identidad. Función id() devuelve identidad.

• Tipo: determina posibles valores y operaciones. Función type() devuelve el tipo. No cambia.

• Valor: que pueden ser mutables e inmutables.

– Tipos mutables: list, dictionary, set y tipos definidos por el usuario.

– Tipos inmutables: int, float, bool, string y tuple.

In [ ]:
```
# 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 [ ]:
```
# 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 [ ]:
```
# 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 [ ]:
```
# 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 [ ]:
```
# 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.

In [69]:
```
# 1
int1 = 3
str1 = "abc"

ans = int1 + str1
print(ans)
print(ans)
```

```
3abc
```

In [70]:
```python
#2
N = int(input("Introduce el número N: "))
a = int(input("Introduce el límite inferior a: "))
b = int(input("Introduce el límite superior b: "))

resultado = (a <= N) and (N < b)

print(resultado)
```

```
True
```

In [72]:
```python
#3

S1 = input("Introduce el string S1: ")
S2 = input("Introduce el string S2: ")
N1 = int(input("Introduce la posición inicial N1: "))
N2 = int(input("Introduce la posición final N2: "))

substring = S1[N1:N2 + 1]

resultado = substring in S2

print(resultado)
```

```
True
```

In [78]:
```python
#4

lista = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5]

sin_duplicados = list(dict.fromkeys(lista))

print(sin_duplicados)
```

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

In [79]:
```python
#5

L = ["rojo", "verde", "azul", "verde"]

s = "verde"
t = "amarillo"

for i in range(len(L)):
    if L[i] == s:
        L[i] = t

print(L)
```

```
['rojo', 'amarillo', 'azul', 'amarillo']
```

In [80]:
```python
#6

tupla = (10, 20, 30, 40)

lista = list(tupla)
```

```
lista[-1] = 99

tupla = tuple(lista)

print(tupla)
```

```
(10, 20, 30, 99)
```

In [81]:
```
#7

lista = [1, 2, 3, 4, 5, 6, 7, 8]

resultado = lista[::-2]

print(resultado)
```

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

In [82]:
```
#8

tupla = (10, 20, 30, 40, 50)

i = 2

nueva_tupla = tupla[:i] + tupla[i+1:]

print(nueva_tupla)
```

```
(10, 20, 40, 50)
```

In [83]:
```
#9

L = [7, 1, 3, 9, 5]

L_ordenada = sorted(L)

n = len(L_ordenada)

if n % 2 != 0:
    mediana = L_ordenada[n // 2]
else:
    mediana = (L_ordenada[n // 2 - 1] + L_ordenada[n // 2]) / 2

print(mediana)
```

```
5
```

In [ ]:
```

```