TipoDatosPython

January 9, 2026

1 ITQ

2 PAO 2025-2026

Nombre: *Israel Pabon*

Fecha: *09/01/2026*

3 0.1 Tercer Modulo- Python , Data Types

Israel Pabon

3.1 1 Comentarios

3.1.1 1.0.1 ¿Qué son?

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

3.1.2 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 comentarios, en lugar de vernos forzados a recurrir a los comentarios para explicar el código.

3.1.3 1.0.3 Tipos de comentarios

3.1.4 Comentarios de una línea

- Texto precedido por '#'.
- Se suele usar para documentar expresiones sencillas.

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

3.1.5 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.

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

3.2 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:

[ ]:
```
4 + 3
```

3.2.1 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

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

3.2.2 Literales compuestos

- Tuplas.
- Listas.
- Diccionarios.
- Conjuntos.

[ ]:
```
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
```

3.2.3 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>d '=' <objeto>

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

[ ]:
```
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))
```

- 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.

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

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

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

- *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.

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

a = 'Pablo García'
print(id(a))

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

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

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

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

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

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

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

[ ]:
```
# # Ojo con los enteros "grandes" [-5, 256]

a = 258
b = 258

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

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

[ ]:
```
a = 400
b = a

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

- Las variables pueden aparecer en expresiones.

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

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

[ ]:
```
c = a + b

print(c)
print(id(c))
```

3.2.4 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!

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

[ ]:
```
print(pow(3,2))
pow = 1             # built-in reasignado
print(pow)
print(pow(3,2))
```

[ ]:
```
def pow(a, b):
    return a + b

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

3.2.5 Asignación múltiple de variables

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

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

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

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

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

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

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

3.3 2.3 Tipos de datos básicos

3.3.1 Bool

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

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

print(a)
print(type(a))

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

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

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

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

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

3.3.2 Números

[ ]:
```
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.
```

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

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

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

3.3.3 Strings

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

[ ]:
```
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
                        #octavo (exclusive). Esta técnica se la conoce como␣
 ↪'slicing'.
print(s[:])             # Todo el string.
print(s + "e")          # Concatenación.
```

3.3.4 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*).

[ ]:
```
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)
```

[ ]:
```
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␣
↪'int'
```

3.3.5 2.5 Operadores

Phyton 3 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 |

[ ]:
```
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)
```

3.3.6 Operadores de comparación

Operador	Descripción
a > b	Suma
a < b	Resta
a == b	División
a != b	División Entera
a >= b	Módulo / Resto
a <= b	Multiplicación

[ ]:
```
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)
```

3.3.7 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.

[ ]:
```
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)
```

3.3.8 Operadores Bitwise / Binarios

Operador	Descripción
a & b	And binario
a	b
a ^ b	Xor binario
~ a	Not binario
a » b	Desplazamiento binario a derecha
a « b	Desplazamiento binario a izquierda

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

3.3.9 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
**=	Exponencial y asignación
&=	And y asignación
’	=’
^=	Xor y asignación
»=	Desplazamiento Derecha y asignación
«=	Desplazamiento Izquierda y asignación

[ ]:
```
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)
```

3.3.10 Operadores de Identidad

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

[ ]:
```
a = 4444
b = a

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

3.3.11 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.

[ ]:
```
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
```

3.3.12 2.6 Entrada de valores

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

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

3.4 3 Tipos de datos compuestos (colecciones)

3.4.1 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*.

3.4.2 Operaciones con listas

- Creación de listas.

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

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

[ ]:
```
# 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
```

[ ]:
```
# Pueden incluso contener objetos más "complejos"
lista_con_funcion = [1, 2, len, pow]
print(lista_con_funcion)
```

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

- Obtención de la longitud de una lista

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

- Acceso a un elemento de una lista.

[ ]:
```
print(letras[2])
print(letras[-5])
print(letras[0])
```

- **Slicing**: obtención de un fragmento de una lista, devuelve una copia de una parte de la lista.

– *Sintaxis: lista [inicio : fin : paso]*

```
[ ]: letras = ['a', 'b', 'c', 'd', 'e']

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

```
[ ]: letras = ['a', 'b', 'c', 'd']

print(letras)
print(id(letras))

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

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

- Añadir un elemento al final de la lista.

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

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

- Insertar en posición.

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

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

- Modificación de la lista (individual).

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

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

- Modificación múltiple usando slicing.

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

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

# print(id(letras))
```

[ ]:
```
# Ojo con la diferencia entre modificación individual y múltiple. Asignación individual de lista crea anidamiento.

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])
```

- Eliminar un elemento.

[ ]:
```
#letras.remove('f')
if 'p' in letras:

letras.remove('p')
#letras.remove('z')

print(letras)
```

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

[ ]:
```
numeros = [1, 2, 3]
print(numeros)
numeros[2] = [10, 20, 30]
print(numeros)

n = numeros[2].pop()
```

```
print(numeros)
print(n)
```

[ ]:
```
# numeros1=10
# print(numeros1)
# print(numeros)
```

[ ]:
```
lista = []
a = lista.pop()
```

- Encontrar índice de un elemento.

[ ]:
```
letras = ['a', 'b', 'c', 'c']

if 'a' in letras:
    print(letras.index('a'))
print(letras)
```

- Concatenar listas.

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

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

[ ]:
```
# Concatenación sin crear una nueva lista
frutas = ['naranja', 'manzana']
print(id(frutas))

frutas.extend(['pera', 'uvas'])
print(frutas)
print(id(frutas))
```

4 Anidar sin crear una nueva lista

frutas = [‘naranja’, ‘manzana’] print(id(frutas))

frutas.append([‘pera’, ‘uvas’]) print(frutas) print(id(frutas))

- Replicar una lista.

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

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

- Copiar una lista

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

- Ordenar una lista.

[ ]:
```
lista = [4,3,8,1]
print(lista)
lista.sort()
print(lista)

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

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

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

- Pertenencia.

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

- Anidamiento.

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

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

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

print(id(a))
```

```
print(id(b))

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

4.0.1 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.

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

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

ref[2] = 64

print(ref)
print(lista)
```

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

[ ]:
```
#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)
```

[ ]:
```
# 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)
```

[ ]:
```
# 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])}")
```

4.0.2 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*.

4.0.3 Operaciones con diccionarios

- Creación de diccionarios.

[ ]:
```
# Creación simple, usando una expresión literal.

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

[ ]:
```
# Creación uniendo dos colecciones.

nombres = ['Pablo', 'Manolo', 'Pepe', 'Juan']
edades = [52, 14, 65]

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

[ ]:
```
# Creación pasando claves y valores a la función 'dict'

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

[ ]:
```
# Creación usando una lista de tuplas de dos elementos.

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

[ ]:
```
# Creación incremental por medio de asignación (como las claves no existen, se␣
 ↪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.

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

[ ]:
```
# 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.

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

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

print(persona)
```

- Añadir un valor a través de la clave.

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

print(persona)
```

- Eliminación de un valor a través de la clave.

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

print(persona)
print(value)
```

- Comprobación de existencia de clave.

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

print('Apellido' in persona)
```

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

[ ]:
```
persona = {'Nombre' : 'Carlos'}

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

# persona['Estatura']

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

- Anidamiento.

[ ]:
```
persona = {
'Trabajos' : ['desarrollador', 'gestor'],
```

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

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

- Métodos *items, keys* y *values*

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

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

[ ]:
```
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)
```

[ ]:
```
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))
```

4.0.4 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.

4.0.5 Operaciones con conjuntos

- Creación de conjuntos.

[ ]:
```
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)
```

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

- Acceso por índice genera error.

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

- Unión, intersección y diferencia.

[ ]:
```
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)
```

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

[ ]:
```
#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.

```
[ ]: 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.

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

- Anidamiento.

```
[ ]: # 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.

```
[ ]: # A través de operador de asignación

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

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

     print(set1)
```

```
[ ]: # 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)
```

[ ]:
```
# A través de métodos 'add' y 'remove'.

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

set1.add('d')
set1.remove('a')
print(set1)
```

4.0.6 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.

4.0.7 ¿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).

4.0.8 Operaciones con tuplas

- Creación de tuplas.

[ ]:
```
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)
```

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

# Creación de tupla de un elemento.
```

```
singleton_tuple = (1,)
print(type(singleton_tuple))
print(singleton_tuple)
```

- Obtención del número de elementos.

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

- *Acceso* por índice.

[ ]:
```
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.

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

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

# print(tuple1)
```

- Contar número de ocurrencias de un elemento.

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

- Encontrar el índice de un elemento.

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

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

[ ]:
```
# Si el elemento no existe, error

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

[ ]:
```
# 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.

[ ]:
```
# 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)
```

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

4.0.9 3.5 Secuencias

- Formalmente, objetos iterables no materializados.
- No son listas. 'list()' para materializarla.
- 'range' o 'enumerate'.

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

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

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

[ ]:
```
# 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)
```

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

[ ]:
```
# 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]}")
```

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

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

[ ]:
```
# 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)
```

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

4.0.10 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.

[ ]:
```
# 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)
```

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

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

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

[ ]:
```
# 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)
```

4.0.11 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.

4.1 4 Soluciones