PAO25_25_02_Python_Datatype

June 25, 2025

1 01PAO25-25 - Python 01

Ing. Elvis Pachacama

Estudiante: Cristopher Santander

2 Comentarios

2.0.1 ¿Qué son?

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

2.0.2 ¿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.

2.0.3 Tipos de comentarios

Comentarios de una línea

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

[2]:
```
# 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.

[3]:
```
def producto(x, y):
    '''
    Esta función recibe dos números como parámetros y devuelve
    como resultado el producto de los mismos.
    '''
    return x * y
```

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]:
```
4+3
```

[4]: 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

[5]:
```
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

[6]:
```
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
```

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

[6]:
```
(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> '=' <objeto>
```

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

[36]:
```
a = 1                # entero
b = 4.0              # coma flotante
c = "VIU"            # 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'>
VIU
<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.

[ ]:
```
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.

[7]:
```
a = 3
print(id(a))

a = 'Pablo García'
print(id(a))

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

```
140730443551592
2451272665104
2451269888976
```

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

[8]:
```
a = 4567
print(id(a))
```

```
2451272463888
```

[9]:
```
b = a
print(id(b))
```

```
2451272463888
```

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

```
2451272453584
2451272455504
```

[14]:
```
a = 25
b = 25

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

```
140730443552296
140730443552296
140730443552296
```

[6]:
```
# Ojo con los enteros "grandes" [-5, 256]
a = 258
b = 258
```

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

```
False
True
2659675632752
2659675628464
2659675632624
```

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

[16]:
```
a = 400
b = a
print(id(a))
print(id(b))
```

```
2451272457648
2451272457648
```

- Las variables pueden aparecer en expresiones.

[17]:
```
a = 3
b = 5
print (a + b)
```

```
8
```

[18]:
```
c = a + b
print(c)
print(id(c))
```

```
8
140730443551752
```

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!

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

```
9
```

[30]:
```
print(pow(3,2))


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

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[30], line 1
----> 1 print(pow(3,2))
      4 #pow = 1  # built-in reasignado
      5 # #print(pow)
      7 print(pow(3,2))

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

[32]:
```
def pow(a, b):
    return a + b
```

Asignación múltiple de variables

[33]:
```
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.

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

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

```
2 1
```

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

c = a
```

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

```
2 1
```

2.3 Tipos de datos básicos

Bool

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

[3]:
```
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.

[4]:
```
a = True
b = True

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

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

```
140730442082880
140730442082880
True
True
```

Números

[7]:
```
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.

[23]:
```
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.

[17]:
```
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.
```

```
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*).

[18]:
```
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
```

[19]:
```
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'
```

```
1.75
1
1
```

2.5 Operadores

Python3 precedencia en operaciones

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

Operatores 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

[21]:
```
x = 3
y = 2

print('x + y = ', x + y)       # Suma: 3 + 2 = 5
print('x - y = ', x - y)       # Resta: 3 - 2 = 1
print('x * y = ', x * y)       # Multiplicación: 3 * 2 = 6
print('x / y = ', x / y)       # División real (float): 3 / 2 = 1.5
print('x // y = ', x // y)     # División entera (int): 3 // 2 = 1
print('x % y = ', x % y)       # Módulo (residuo): 3 % 2 = 1
print('x ** y = ', x ** y)     # Exponenciación: 3 ** 2 = 9
```

```
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

[22]:
```
x = 10
y = 12

print('x > y  es ', x > y)    # False → 10 no es mayor que 12
print('x < y  es ', x < y)    # True  → 10 sí es menor que 12
print('x == y es ', x == y)   # False → no son iguales
print('x != y es ', x != y)   # True  → sí son diferentes
print('x >= y es ', x >= y)   # False → 10 no es mayor ni igual a 12
print('x <= y es ', x <= y)   # True  → 10 sí es menor o igual a 12
```

```
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

Enlace a Tablas de Verdad.

[24]:
```
x = True
y = False

print('x and y es :', x and y)  # False → ambos deben ser True para que el␣
 ↪resultado sea True
print('x or y es  :', x or y)   # True  → basta con que uno sea True
print('x xor y es :', x ^ y)    # True  → operador XOR (exclusivo): True solo␣
 ↪si los valores son distintos
print('not x es   :', not x)    # False → invierte el valor de 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

[25]:
```
x = 0b01100110  # Valor binario equivalente a 102 en decimal
y = 0b00110011  # Valor binario equivalente a 51 en decimal

print("Not x = " + bin(~x))       # Operación bit a bit NOT: invierte todos los␣
 ↪bits → -0b1100111 (resultado en complemento a dos)
print("x and y = " + bin(x & y))  # AND bit a bit: solo deja en 1 los bits que␣
 ↪estén en 1 en ambos operandos → 0b100010
print("x or y = " + bin(x | y))   # OR bit a bit: pone en 1 los bits que estén␣
 ↪en 1 en al menos uno de los operandos → 0b1110111
print("x xor y = " + bin(x ^ y))  # XOR bit a bit: pone en 1 los bits que son␣
 ↪diferentes entre x e y → 0b1010101
print("x << 2 = " + bin(x << 2))  # Desplazamiento a la izquierda: mueve los␣
 ↪bits dos posiciones a la izquierda, rellenando con ceros → 0b110011000␣
 ↪(equivale a multiplicar por 4)
print("x >> 2 = " + bin(x >> 2))  # Desplazamiento a la derecha: mueve los bits␣
 ↪dos posiciones a la derecha, descartando los últimos → 0b11001 (equivale a␣
 ↪dividir entre 4 sin decimales)
```

```
Not x = -0b1100111
x and y = 0b100010
x or y = 0b1110111
x xor y = 0b1010101
x << 2 = 0b110011000
x >> 2 = 0b11001
```

Operadores de Asignación

Operador	Desc
=	Asignación
+=	Suma y asignación

Operador	Desc
-=	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

[27]:
```
a = 5
a *= 3   # Multiplica a por 3 y reasigna el resultado a a → a = 15
a += 1   # Suma 1 al valor actual de a → a = 16
         # En Python no existen los operadores a++ ni ++a como en C/C++
print(a) # Imprime: 16

b = 6
b -= 2   # Resta 2 al valor actual de b → b = 4
print(b) # Imprime: 4
```

```
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

[28]:
```
a = 4444       # Se crea una variable a con valor entero 4444
b = a          # b apunta al mismo objeto que a en memoria (no es una copia, es␣
 ↪la misma referencia)

print(a is b)       # True → a y b apuntan al mismo objeto en memoria
print(a is not b)   # False → ya que no son objetos distintos, sino el mismo
```

```
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

[29]:
```
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

[30]:
```
valor = input("Inserte valor:")
print(valor)
```

```
Inserte valor: 5
```

```
5
```

[31]:
```
grados_c = input("Conversión de grados a fahrenheit, inserte un valor: ")
# Solicita al usuario que escriba una temperatura en grados Celsius. El␣
 ↪resultado será un string.

print(f"Grados F: {1.8 * int(grados_c) + 32}")
# Convierte ese string a entero con int()
# Aplica la fórmula de conversión: (°C × 1.8) + 32
# Muestra el resultado con un f-string (formateo elegante)
```

```
Conversión de grados a fahrenheit, inserte un valor:  8
```

```
Grados F: 46.4
```

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.

[33]:
```
letras = ['a', 'b', 'c', 'd']           # Lista explícita de caracteres (strings␣
 ↪de un solo carácter)
palabras = 'Hola mundo'.split()         # Divide el string por espacios →␣
 ↪['Hola', 'mundo']
numeros = list(range(5))                # Crea una lista de números del 0 al 4 →␣
 ↪[0, 1, 2, 3, 4]

print(letras)                           # ['a', 'b', 'c', 'd']
print(palabras)                         # ['Hola', 'mundo']
print(numeros)                          # [0, 1, 2, 3, 4]
print(type(numeros))                    # <class 'list'>
```

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

[ ]:
```

```