PAO25_25_02_Python_Datatype

June 25, 2025

1 01PAO25-25 - Python, Data Types

Alan Fernando Solano

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.

[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.

[2]:
```
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:

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

[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

[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

[5]:
```
print([1, 2, 3, 3, 6, 1])                          # lista - mutable
print({'Nombre' : 'John Doe', "edad": 30})  # Diccionario - mutable
print({1, 2, 3, 3, 4, 2})                          # Conjunto - mutable
print((4, 5, 6))                                  # tupla - inmutable
2, 4
```

```
[1, 2, 3, 3, 6, 1]
{'Nombre': 'John Doe', 'edad': 30}
{1, 2, 3, 4}
(4, 5, 6)
```

[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> '=' <objeto>
```

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

```
5
```

[7]:
```
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.

[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.

[9]:
```
a = 3
print(id(a))

a = 'Pablo García'
print(id(a))

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

```
140714498446184
2649474398128
2649455049104
```

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

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

```
2649472067120
```

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

```
2649472067120
```

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

```
2649472062992
```

[13]:
```
a = 25
b = 25
c=25

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

```
140714498446888
140714498446888
140714498446888
140714498446888
```

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

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

```
2649472067696
2649472061488
2649472065392
```

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

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

```
2649472068048
2649472068048
```

- Las variables pueden aparecer en expresiones.

[16]:
```
a = 3
b = 5
c=a+b
print (a + b)
```

```
8
```

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

```
8
140714498446344
```

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!

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

```
27
```

[19]:
```
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[19], line 6
      3 pow = 1  # built-in reasignado
      4 print(pow)
----> 6 print(pow(3,2))

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

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

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.

[20]:
```
a = 1
b = 2

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

```
2 1
```

[21]:
```
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'.

[22]:
```
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.

[23]:
```
a = True
b = True

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

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

```
140714496977472
140714496977472
True
True
```

Números

[24]:
```
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.

[25]:
```
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.

[26]:
```
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*).

[27]:
```
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
```

[28]:
```
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

[29]:
```
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

Operador	Desc
a <= b	Menor o Igual

[30]:
```
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

Enlace a Tablas de Verdad.

[31]:
```
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

[32]:
```
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))
```

```
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
-=	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

[33]:
```
a = 5
a *= 3   # a = a * 3
a += 1   # No existe a++, ni ++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

[34]:
```
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

[35]:
```
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

[36]:
```
valor = input("Inserte valor:")
print(valor)
```

```
Inserte valor: 17
```

```
17
```

[42]:
```
grados_c = input("Conversión de grados a fahrenheit, inserte un valor: ")
print(f"Grados F: {1.8 * int(grados_c) + 32}")
```

```
Conversión de grados a fahrenheit, inserte un valor:  60
```

```
Grados F: 140.0
```

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.

[43]:
```
letras = ['a', 'b', 'c', 'd']
# LA función split divide cadena de textos en sublistas
palabras = 'Hola mundo'.split()
numeros = list(range(5))

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

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

[44]:
```
# Pueden contener elementos arbitrarios / heterogeneos
mezcla = [1, 3.4, 'a', None, False]
print(mezcla)
print(len(mezcla))
```

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

[45]:
```
# Pueden incluso contener objetos más "complejos"
lista_con_funcion = [1, 2, len]
print(lista_con_funcion)
```

```
[1, 2, <built-in function len>]
```

[46]:
```
Pueden contener duplicados
lista_con_duplicados = [1, 2, 3, 3, 3, 4]
print(lista_con_duplicados)
```

```
  Cell In[46], line 1
    Pueden contener duplicados
           ^
SyntaxError: invalid syntax
```

- Acceso a un elemento de una lista

[47]:
```
print(letras[2])
print(letras[-1])
```

```
c
d
```

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

[48]:
```
letras = ['a', 'b', 'c', 'd']

print(letras[1:3])
print(letras[:1])
#Toma todos los elementos desde el inicio de la lista hasta el penúltimo (sin␣
 ↪incluir el último)
print(letras[:-1])
print(letras[2:])
print(letras[:])
print(letras[::2])
```

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

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

[49]:
```
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']
2649475173888
['a', 'b', 'c', 'd']
2649475174848
['a', 'b', 'c', 'd']
2649475183616
```

- Añadir un elemento al final de la lista

[50]:
```
letras.append('e')
print(letras)
print(id(letras))
```

```
['a', 'b', 'c', 'd', 'e']
2649475173888
```

[51]:
```
letras += 'e'
print(letras)
print(id(letras))
```

```
['a', 'b', 'c', 'd', 'e', 'e']
2649475173888
```

- Insertar en posición.

[52]:
```
letras.insert(1,'g')
print(letras)
print(id(letras))
```

```
['a', 'g', 'b', 'c', 'd', 'e', 'e']
2649475173888
```

- Modificación de la lista (individual).

[53]:
```
letras[5] = 'f'
print(letras)
print(id(letras))
```

```
['a', 'g', 'b', 'c', 'd', 'f', 'e']
2649475173888
```

[54]:
```
# index tiene que estar en rango
letras[8] = 'r'
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[54], line 2
      1 # index tiene que estar en rango
----> 2 letras[8] = 'r'

IndexError: list assignment index out of range
```

- Modificación múltiple usando slicing.

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

[ ]:
```
letras[0:6:2] = ['z', 'z', 'z']
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][2])

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

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

- Eliminar un elemento.

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

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

[55]:
```
numeros = [1, 2, 3]
numeros[2] = [10, 20, 30]
print(numeros)

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

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

[56]:
```
lista = []
a = lista.pop()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[56], line 2
      1 lista = []
----> 2 a = lista.pop()

IndexError: pop from empty list
```

- Encontrar índice de un elemento.

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

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

- Concatenar listas.

[57]:
```
lacteos = ['queso', 'leche']
frutas = ['naranja', 'manzana']
print(id(lacteos))
print(id(frutas))

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

```
2649472733888
2649472737728
```

```
2649472730048
['queso', 'leche', 'naranja', 'manzana']
```

[58]:
```
Concatenación sin crear una nueva lista
frutas = ['naranja', 'manzana']
print(id(frutas))
frutas.extend(['pera', 'uvas'])
print(frutas)
print(id(frutas))
```

```
  Cell In[58], line 1
    Concatenación sin crear una nueva lista
                  ^
SyntaxError: invalid syntax
```

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

- Replicar una lista.

[59]:
```
lacteos = ['queso', 'leche']
print(lacteos * 3)
print(id(lacteos))

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

```
['queso', 'leche', 'queso', 'leche', 'queso', 'leche']
2649478043264
['queso', 'leche', 'queso', 'leche', 'queso', 'leche']
2649478290048
```

- Copiar una lista

[60]:
```
frutas2 = frutas.copy()
frutas2 = frutas[:]
print(frutas2)
print('id frutas = ' + str(id(frutas)))
print('id frutas2 = ' + str(id(frutas2)))
```

```
['naranja', 'manzana']
id frutas = 2649472737728
id frutas2 = 2649476395200
```

- Ordenar una lista.

[61]:
```
lista = [4,3,8,1]
lista.sort()
print(lista)

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

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

[62]:
```
# Los elementos deben ser comparables para poderse ordenar
lista = [1, 'a']
lista.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[62], 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'
```

[63]:
```
compra = ['Huevos','Pan','Leche']
print(sorted(compra))
print(compra)
```

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

- Pertenencia.

[64]:
```
lista = [1, 2, 3, 4]
print(1 in lista)
print(5 in lista)
```

```
True
False
```

- Anidamiento.

[65]:
```
letras = ['a', 'b', 'c', ['x', 'y', ['i', 'j', 'k']]]
print(letras[0])
print(letras[3][0])
print(letras[3][2][0])
```

```
a
x
```

```
i
```

[ ]:

[ ]: