


3 Tipos de datos básicos y operadores.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

3.1 Tipos y operadores

En Python se tienen tres tipos de datos básicos principales:

Tipo	Ejemplo
Númerico	13, 3.1416, 1+5j
Cadena	"Frida", "Diego"
Lógico	True, False

3.2 Tipos numéricos

En Python se tienen tres tipos de datos numéricos: 1. Enteros 2. Reales 3. Complejos

A continuación se realiza una descripción de estos tipos numéricos. Más información se puede encontrar aquí: [Numeric types](#).

1. Enteros

Son aquellos que carecen de parte decimal. Para definir un entero hacemos lo siguiente:

```
entero = 13
```

Cuando se ejecuta la celda anterior, se crea el objeto `13` cuyo nombre es `entero`. Podemos imprimir el valor de `entero` y su tipo como sigue:

```
print(entero)
print(type(entero))
```

```
13
<class 'int'>
```

Es posible obtener más información del tipo `int` usando la biblioteca `sys`:

```
import sys
sys.int_info
```

```
sys.int_info(bits_per_digit=30, sizeof_digit=4, default_max_str_digits=4300,
str_digits_check_threshold=640)
```

2. Reales

Son aquellos que tienen una parte decimal. Para definir un número real (flotante) se hace como sigue:

```
pi = 3.141592
```

Cuando se ejecuta la celda anterior, se crea el objeto `3.141592` cuyo nombre es `pi`. Podemos imprimir el valor de `pi` y su tipo como sigue:

```
print(pi)
print(type(pi))
```

```
3.141592
<class 'float'>
```

```
# para obtener más información:
sys.float_info
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

3. Complejos

Son aquellos que tienen una parte real y una parte imaginaria, y ambas partes son números reales. Para definir un número complejo se hace como sigue:

```
complejo = 12 + 5j # La parte imaginaria lleva una j al final
```

Cuando se ejecuta la celda anterior, se crea el objeto `12 + 5j` cuyo nombre es `complejo`. En este caso, el contenido de `complejo` tiene dos partes: la real y la imaginaria. Podemos imprimir el valor de `complejo` y su tipo como sigue:

```
print(complejo)
print(type(complejo))
```

```
(12+5j)
<class 'complex'>
```

```
complejo.imag # accedemos a la parte imaginaria
```

5.0

```
complejo.real # accedemos a la parte real
```

12.0

```
complejo.conjugate() # calculamos el conjugado del número complejo.
```

(12-5j)

Nota: observa que hemos aplicado el método `conjugate()` al objeto `complejo`, esto es posible debido a que existe la clase `<class 'complex'>` en Python, y en ella se definen atributos y métodos para los objetos de esta clase. Más acerca de programación orientada a objetos la puedes ver en esta sección XXX.

3.2.1 Operadores aritméticos

Para los tipos numéricos descritos antes, existen operaciones aritméticas que se pueden aplicar sobre ellos. Veamos:

```
# Suma
1 + 2
```

```
# Resta
5 - 32
```

```
# Multiplicación
3 * 3
```

```
# División
3 / 2
```

```
# Potencia
81 ** (1/2)
```

3.2.2 Precedencia de operadores.

La aplicación de los operadores tiene cierta precedencia que está definida en cada implementación del lenguaje de programación. La siguiente tabla muestra el orden en que se aplicarán los operadores en una expresión.

Nivel	Categoría	Operadores
7	exponenciación	<code>**</code>
6	multiplicación	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>

Nivel	Categoría	Operadores
5	adición	<code>+</code> , <code>-</code>
4	relacional	<code>==</code> , <code>!=</code> , <code><=</code> , <code>>=</code> , <code>></code> , <code><</code>
3	logicos	<code>not</code>
2	logicos	<code>and</code>
1	logicos	<code>or</code>

Como se puede ver, siempre se aplican primero los operadores aritméticos (niveles 7,6, y 5), luego los relacionales (nivel 4) y finalmente los lógicos (3, 2 y 1).

Más acerca de este tema se puede ver aquí: [Operator precedence](#).

A continuación se muestran ejemplos de operaciones aritméticas en donde se resalta est precedencia:

```
# Precedencia de operaciones: primero se realiza la multiplicación
1 + 2 * 3 + 4
```

11

```
# Es posible usar paréntesis para modificar la precedencia: primero la suma
(1 + 2) * (3 + 4)
```

21

```
# ¿Puedes explicar el resultado de acuerdo con la precedencia
# descrita en la tabla anterior?
6/2*(2+1)
```

9.0

3.2.3 Operaciones entre tipos diferentes

Es posible combinar operaciones entre tipos de números diferentes. Lo que Python hará es promover cada número al tipo más sofisticado, siendo el orden de sofisticación, de menos a más, como sigue: `int`, `float`, `complex`.

```
a = 1 # un entero
b = 2 * 3j # un complejo
a + b # resultará en un complejo
```

3.2.4 Operadores de asignación

Existen varios operadores para realizar asignaciones: `=`, `+=`, `-=`, `*=`, `/=`, `**=`, `%=`. La forma de uso de estos operadores se muestra en los siguientes ejemplos:

```
etiqueta = 1.0
suma = 1.0
suma += etiqueta # Equivalente a : suma = suma + etiqueta
print(suma)
```

```
etiqueta = 4
resta = 16
resta -= etiqueta # Equivalente a : resta = resta - etiqueta
print(resta)
```

```
etiqueta = 2
mult = 12
mult *= etiqueta # Equivalente a : mult = mult * etiqueta
print(mult)
```

```
etiqueta = 5
div = 50
div /= etiqueta # Equivalente a : divide = divide / etiqueta
print(div)
```

```
etiqueta = 2
pot = 3
pot **= etiqueta # Equivalente a : pot = pot ** etiqueta
print(pot)
```

```
etiqueta = 5
modulo = 50
modulo %= etiqueta # Equivalente a : modulo = modulo % etiqueta
print(modulo)
```

3.3 Tipos lógicos

Es un tipo utilizado para realizar operaciones lógicas y puede tomar dos valores: `True` o `False`.

```
bandera = True
print(type(bandera))
```

3.3.1 Operadores relacionales

Cuando se aplica un operador relacional a dos expresiones, se realiza una comparación entre dichas expresiones y se obtiene como resultado un tipo lógico. **True** o **False**.

Los operadores relacionales que se pueden usar son: **==**, **!=**, **<=**, **>=**, **>**, **<**. A continuación se muestran algunos ejemplos:

```
35 > 562 # ¿Es 35 mayor que 562?
```

False

```
32 >= 21 # ¿Es 32 mayor o igual que 21?
```

True

```
12 < 34 # ¿Es 12 menor que 34?
```

True

```
12 <= 25 # ¿Es 12 menor o igual que 25?
```

True

```
5 == 5 # ¿Es 5 igual a 5?
```

True

```
23 != 23 # ¿Es 23 diferente de 23?
```

False

```
'aaa' == 'aaa' # Se pueden comparar otros tipos de datos
```

True

```
5 > len('5')
```

True

3.3.2 Operaciones lógicas.

Los operadores lógicos que se pueden usar son: **not** , **and** y **or** . Veamos algunos ejemplos

```
(5 < 32) and (63 > 32)
```

True

Debido a la precedencia de operadores, no son necesarios los paréntesis en la operaciones relacionales de la expresión anterior (véase tabla ...):

```
5 < 32 and 63 > 32
```

True

Aunque a veces el uso de paréntesis hace la lectura del código más clara:

```
(2.32 < 21) and (23 > 63)
```

False

```
(32 == 32) or (5 < 31)
```

True

```
(32 == 21) or (31 < 5)
```

False

```
not True
```

False

```
not (32 != 32)
```

True

3.3.2.1 Comparación entre números flotantes.

La comparación entre números de tipo flotante debe realizarse con cuidado, veamos el siguiente ejemplo:

```
(0.4 - 0.3) == 0.1
```

False

El cálculo a mano de $(0.4 - 0.3)$ da como resultado 0.1 ; pero en una computadora este cálculo es aproximado y depende de las características del hardware (exponente, mantisa, base, véase). En Python el resultado de la operación $(0.4 - 0.3)$ es diferente de 0.1 , veamos:

```
print(0.4 - 0.3)
```

```
0.10000000000000003
```

Python ofrece herramientas que permiten realizar una mejor comparación entre números de tipo flotante. Por ejemplo la biblioteca `math` contiene la función `isclose(a, b)` en donde se puede definir una tolerancia mínima para que las dos expresiones, `a` y `b` se consideren iguales (*cercanas*), por ejemplo:

```
import math
math.isclose((0.4 - 0.3), 0.1)
```

```
True
```

Se recomienda revisar el manual de [math.isclose\(\)](#) y el de [numpy.isclose\(\)](#) para comparación de arreglos con elementos de tipo flotante.

3.4 Fuertemente Tipado.

Python es fuertemente tipado, lo que significa que el tipo de un objeto no puede cambiar repentinamente; se debe realizar una conversión explícita si se desea cambiar el tipo de un objeto.

Esta característica también impide que se realicen operaciones entre tipos no compatibles.

Veamos unos ejemplos:

```
lógico = True
real    = 220.0
entero  = 284
complejo = 1+1j
cadena  = 'numeros hermanos'
```

```
lógico + real # Los tipos son compatibles
```

```
221.0
```

```
lógico + complejo # Los tipos son compatibles
```

```
(2+1j)
```

```
cadena + real # Los tipos no son compatibles
```


TypeError: can only concatenate str (not "float") to str

3.5 Conversión entre tipos (*casting*)

Es posible transformar un tipo en otro tipo compatible; a esta operación se lo conoce como *casting*.

3.5.1 Función `int()`

Transforma objetos en enteros, siempre y cuando haya compatibilidad.

```
cadena = '1000'  
print(type(cadena))  
entero = int(cadena)  
print(type(entero))  
print(entero)
```

```
<class 'str'>  
<class 'int'>  
1000
```

```
flotante = 3.141592  
entero = int(flotante) # Trunca la parte decimal  
print(entero)
```

3

```
complejo = 4-4j  
entero = int(complejo) # Tipos NO COMPATIBLES
```

TypeError: int() argument must be a string, a bytes-like object or a real number, not 'complex'

```
entero = int(True)  
print(entero)
```

1

```
print(1 == True)
```

True

Función `str()`

Transforma objetos en cadenas, siempre y cuando haya compatibilidad.

```
entero = 1000
print(type(entero))
cadena = str(entero)
print(type(cadena))
print(cadena)
```

```
<class 'int'>
<class 'str'>
1000
```

```
complejo = 5+1j
print(complejo)
print(type(complejo))
cadena = str(complejo)
print(cadena)
print(type(cadena))
```

```
(5+1j)
<class 'complex'>
(5+1j)
<class 'str'>
```

Función `float()`

Transforma objetos en flotantes, siempre y cuando haya compatibilidad.

```
cadena = '3.141592'
print(cadena)
print(type(cadena))
real = float(cadena)
print(real)
print(type(real))
```

```
3.141592
<class 'str'>
3.141592
<class 'float'>
```

```
float(33)
```

```
33.0
```

```
float(False)
```

```
0.0
```

```
float(3+3j) # NO hay compatibilidad
```

`TypeError: float() argument must be a string or a real number, not 'complex'`

En general, si existe el tipo `<class 'MiClase'>`, donde `MiClase` puede ser un tipo de dato definido dentro de Python, alguna biblioteca o creada por el usuario, es posible realizar el *casting* del objeto `a` al tipo `<class 'MiClase'>` haciendo: `MiClase(a)` siempre y cuando haya compatibilidad.

3.6 Constantes

Python contiene una serie de constantes integradas a las que no se les puede cambiar su valor.

Más detalles se pueden encontrar en: [Built-in Constants](#)

Las principales constantes son las siguientes:

- `False`: de tipo Booleano.
- `True`: de tipo Booleano.
- `None`: El único valor para el tipo `NoneType`. Es usado frecuentemente para representar la ausencia de un valor, por ejemplo cuando no se pasa un argumento a una función.
- `NotImplemented`: es un valor especial que es regresado por métodos binarios especiales (por ejemplo `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) para indicar que la operación no está implementada con respecto a otro tipo.
- `Ellipsis`: equivalente a `...`, es un valor especial usado mayormente en conjunción con la sintaxis de *slicing* de arreglos.
- `__debug__`: Esta constante es verdadera si Python no se inició con la opción `-O`.

Las siguiente constantes son usadas dentro del intérprete interactivo (no se pueden usar dentro de programas ejecutados fuera del intérprete).

- `quit` (code=None)
- `exit` (code=None)
- `copyright`
- `credits`
- `license`

```
copyright()
```

Copyright (c) 2001–2023 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995–2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991–1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.

```
license()
```

A. HISTORY OF THE SOFTWARE

=====

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <https://www.cwi.nl>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <https://www.cnri.reston.va.us>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see <https://opensource.org> for the Open Source Definition). Historically, most, but not all, Python

Hit Return for more, or q (and Return) to quit: q