Unidad 2.1

Datos numéricos

1. Introducción

Python trabaja con diferentes tipos numéricos como vimos en la lección anterior:

- Enteros (int): representa los números enteros, tanto positivos como negativos, sin parte decimal.
- Reales (float): representa números decimales.
- Complejos (complex): números con parte real e imaginaria. Suele ser más usado en cálculo, análisis de señales, procesamiento de imágenes…

En este tema recorreremos estos datos, viendo sus diferentes funciones, posibles operaciones, conversiones…

1.1. Enteros

Los enteros representan valores numéricos sin decimales, es decir, números que pueden ser positivos, negativos o cero. Este tipo de dato es fundamental, ya que se emplea en una gran variedad de operaciones, desde conteos y cálculos aritméticos hasta la representación de datos en estructuras más complejas.

Python maneja los enteros de manera muy eficiente y permite trabajar con números tan grandes como sea necesario, sin limitar su tamaño por el sistema operativo o el hardware, siempre que la memoria lo permita. Esto lo diferencia de otros lenguajes de programación, donde los enteros suelen tener un rango limitado.

Para crear enteros tan solo hay que asignar su valor a una variable y python interpretará que es un entero:

```
positivo = 7
cero = 0
negativo = -3

print(type(positivo))
print(type(cero))
print(type(negativo))
```

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

También se pueden obtener convirtiendo otros tipos de datos en enteros. Aquí están las formas más comunes:

Uso de la Función int() para Conversiones

- Podemos usar int() para convertir otros tipos de datos a enteros, siempre que la conversión sea válida.

a) Convertir un Número Flotante a Entero

- La función int() elimina la parte decimal, redondeando hacia cero.

```
x = int(3.9)  # x será 3
y = int(-2.8) # y será -2

print(x,y)
```

```
3 -2
```

b) Convertir una Cadena de Texto a Entero

- Si el texto representa un número entero, int() lo convertirá.

```
x = int("123")    # x será 123
y = int("-456")   # y será -456

print(x,y)
```

```
123 -456
```

- Si la cadena contiene texto no numérico, int() generará un error.

```
z = int("Prueba")

print(z)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[6], line 1
----> 1 z = int("Prueba")
      3 print(z)

ValueError: invalid literal for int() with base 10: 'Prueba'
```

c) Convertir un Número en Base Diferente (Binaria, Octal, Hexadecimal)

- int() puede aceptar un segundo argumento para especificar la base (de 2 a 36) del número.

```
x = int("1010", 2)    # x será 10 en base 10 (a partir del binario "1010")
y = int("7F", 16)     # y será 127 en base 10 (a partir del hexadecimal "7F")

print(x,y)
```

```
10 127
```

3. **Crear Enteros con Notación Científica**

- En Python, los números en notación científica siempre se interpretan como flotantes. Para crear un entero a partir de ellos, podemos convertirlos usando int().

```
x = int(5e3)  # x será 5000 (5 * 10^3)

print(x)
```

```
5000
```

4. **Uso de la Función math.floor() o math.ceil() (Redondeo de Flotantes a Entero)**

- Si estamos trabajando con números decimales y queremos redondearlos al entero más cercano, podemos usar math.floor() (redondeo hacia abajo) o math.ceil() (redondeo hacia arriba).

```python
import math

x = math.floor(3.7)  # x será 3
y = math.ceil(3.2)   # y será 4

print(x,y)
```

```
3 4
```

5. **Uso de Redondeo con round()**
   - La función round() redondea un número flotante al entero más cercano.

```python
x = round(3.5)  # x será 4
y = round(2.4)  # y será 2

print(x,y)
```

```
4 2
```

1.2. Reales (Float, números con decimales)

Los *float* en Python representan valores numéricos con decimales. Los flotantes son esenciales cuando necesitamos realizar cálculos más precisos que usen fracciones o decimales, como en ciencias, finanzas, gráficos y simulaciones, donde la precisión es clave.

Los números flotantes se crean automáticamente cuando se usa un decimal en un número, como 3.14 o 0.001. Además, los flotantes permiten representar números en notación científica, lo que facilita trabajar con valores extremadamente grandes o pequeños (por ejemplo, 6.022e23 para expresar el número de Avogadro).

Python maneja los flotantes de manera eficiente, pero debido a la forma en que se representan internamente (basándose en el estándar IEEE 754 de precisión doble), tienen algunas limitaciones inherentes:

1. Imprecisión en Representaciones Decimales

Los números decimales en base 10 no siempre se pueden representar de manera exacta en binario (base 2), que es el sistema usado por las computadoras. Por ejemplo, el número decimal 0.1 no tiene una representación binaria exacta y se convierte internamente en un número ligeramente diferente.

```python
print(0.1 + 0.2)
```

```
0.30000000000000004
```

Debido a esta imprecisión, Python puede producir resultados inesperados al redondear flotantes o al realizar operaciones matemáticas acumulativas. Estos errores pueden ser mínimos, pero se vuelven más significativos al realizar cálculos en bucles grandes o en aplicaciones donde la precisión es crucial, como cálculos financieros. A veces es necesario redondear para corregir la salida.

3. Pérdida de Precisión en Operaciones con Números Muy Grandes o Muy Pequeños.

Si realizamos operaciones entre números flotantes de magnitudes muy distintas, es posible que el número más pequeño pierda precisión o incluso sea ignorado en la operación, debido a los límites de precisión del sistema de flotantes. Por ejemplo, sumar 1e16 y 1e-16 puede resultar en 1e16 porque el número más pequeño se "pierde" en la operación.

```
print(1e16 + 1e-16)
```

```
1e+16
```

4. Limitaciones de Rango de los Flotantes.

Aunque Python permite trabajar con números flotantes de grandes rangos, aún tiene límites. Un número flotante que supere 1.8e308 se considera "infinito" (inf) en Python, y cualquier operación adicional que lo incremente mantendrá este resultado como inf. Esto significa que cálculos que involucran números extremadamente grandes pueden resultar en resultados de inf o -inf, afectando la precisión del cálculo.

```
print(1e308 * 10)
```

```
inf
```

Todo esto hace que tengamos que tener especial cuidado tanto en las operaciones como en las comparaciones con números decimales, ya que si no podremos tener errores que cambien el funcionamiento de nuestro código.

```
x = 0.1 + 0.2
print(x == 0.3)
```

```
False
```

Tenemos dos módulos que nos pueden ayudar a ser más precisos con los cálculos:

El Módulo *decimal*: Permite una mayor precisión y control sobre el redondeo, ideal para aplicaciones financieras.

El Módulo *fractions*: Permite representar números racionales como fracciones exactas, lo cual elimina algunos problemas de precisión al representar ciertos valores.

1. Uso de la Función float() para Conversiones

- Podemos usar float() para convertir otros tipos de datos a enteros, siempre que la conversión sea válida.

a) Convertir un Entero a flotante

- La función float() en números enteros añade .0 para convertirlo en flotante.

```
x = float(3)
y = float(-7)

print(x,y)
```

```
3.0 -7.0
```

b) Convertir una Cadena de Texto a flotante

- Si el texto representa un número entero, float() lo convertirá.

```
x = float("3.1")
y = float("7")

print(x,y)
```

```
3.1 7.0
```

- Si la cadena contiene texto no numérico, float() generará un error.

2. **Crear flotantes con Notación Científica**
   - En Python, los números en notación científica siempre se interpretan como flotantes, pero podemos castearlos con float() también y no dará error.

3. **Los valores booleanos True y False se pueden convertir a flotantes.**
   - Donde True se convierte a 1.0 y False se convierte a 0.0.

```
x = float(True)  # Resultado: 1.0
y = float(False) # Resultado: 0.0

print(x,y)
```

```
1.0 0.0
```

1.3. Complejos

De los números complejos, solo veremos sus posibles formas de construcción. Podemos crearlos asignando a una variable un valor de la forma *a + bj* o utilizando la función *complex(a,b)*, siendo *a* y *b* números enteros o flotantes.

En este curso, no profundizaremos en el tema de números complejos, ya que su uso está principalmente relacionado con áreas avanzadas de matemáticas, ingeniería y física. Aunque Python ofrece soporte nativo para los números complejos, esta funcionalidad es más relevante para quienes trabajen en estas áreas especializadas. Es importante señalar que tienen funciones propias que serán muy útiles como son *.real*, *.imag*, *.conjugate* y más.

2. Operadores aritméticos

Python permite utilizar una variedad de operadores aritméticos con los tipos de datos numéricos, como enteros (int), flotantes (float), y complejos (complex). Estos operadores nos ayudan a realizar operaciones matemáticas básicas y avanzadas de manera sencilla. A continuación, se explican los operadores aritméticos más comunes y su funcionamiento.

1. **Suma** (+)
2. **Resta** (-)
3. **Multiplicación** (*)
4. **División** (/)

5. **División Entera** (//). Realiza la división y devuelve solo la parte entera del resultado. Muy útil para obtener cocientes sin decimales.
6. **Módulo** (%). Devuelve el resto de la división entre dos números. Puede usarse para determinar si un número es divisible por otro.
7. **Potencia** (**). Eleva un número a la potencia de otro.

En general, cuando hay operaciones entre dos tipos diferentes de datos, el resultado será del tipo más “complejo”, siendo el más sencillo int, después float, y después complex.

Orden de Operaciones
En Python, el orden de las operaciones aritméticas sigue las reglas estándar de jerarquía:

Paréntesis ()
Potencias **
Multiplicación *, División /, División Entera //, y Módulo %
Suma + y Resta -

Funciones predefinidas que trabajan con números:

- abs(x): Devuelve al valor absoluto de un número.
- divmod(x,y): Devuelve una tupla con dos la división entera y el módulo (resto de la división) entre los números proporcionados.
- hex(x): convierte el número a una cadena en representación hexadecimal.
- bin(x): convierte el número a una cadena en representación binaria.
- pow(x,y): Devuelve la potencia de la base x elevado al exponente y. Similar al operador**`.
- round(x,[y]): Devuelve un número real que es el redondeo del número recibido como parámetro. Podemos indicar un parámetro opcional que indica el número de decimales en el redondeo (en ese caso, el resultado que nos devuelve es de tipo float).

Otras funciones
Además de las comentadas anteriormente, Python cuenta con el módulo *math* que nos proporciona otras funciones interesantes para trabajar con números. Algunos ejemplos son:

- math.sin(x), math.cos(x), math.tan(x): Devuelve el seno, coseno o tangente de x en radianes.
- math.log(x, y): Devuelve el logaritmo de x en la base y.
- math.exp(x): Devuelve e elevado a la potencia de x.
- math.sqrt(x): Devuelve la raiz cuadrada de x.

Si queréis ver más funciones existentes podéis consultarlas en la documentación oficial de la librería (Aquí).