1 INTRODUCCIÓN A PYTHON: TIPOS DE DATOS, VARIABLES, OPERADORES Y EXPRESIONES

1.1 Expresiones Matemáticas

Antes de comenzar a trabajar en Python a nivel general, es necesario aprender a realizar operaciones aritméticas básicas.

1.1.1 Expresiones Binarias

La notación para realizar operaciones aritméticas básicas **binarias** es la siguiente:

Número –> Operador –> Número

[ ]:
```
# Suma
5 + 4
```

[ ]:
```
9
```

[ ]:
```
# Suma
8.0 + 3
```

[ ]:
```
11.0
```

[ ]:
```
# Resta
3 - 7
```

[ ]:
```
-4
```

[ ]:
```
# Resta
10 - 5.4
```

[ ]:
```
4.6
```

Los ejemplos anteriores se conocer como **notación infija**, y es la que se debe utilizar cada vez que se escriban operaciones matemáticas.

Hay que considerar que para realizar una `multiplicación` se debe utilizar el operador * (**asterisco**) y para realizar una `división` se debe utilizar el operador / (**slash**)

[ ]:
```
# Multiplicación
2*4
```

[ ]:
```
8
```

[ ]:
```
# Multiplicación
2.5*5
```

[ ]:
```
12.5
```

[ ]:
```
# División
10/5
```

[ ]: 2.0

[ ]:
```
# División
12/8
```

[ ]: 1.5

Además de las 4 operaciones básicas, Python provee operaciones que pueden ser de utilidad:

- **Potencia** (**)
- **Resto o módulo** (%)
- **División parte entera** (//)

[ ]:
```
# Potencia
2**4
```

[ ]: 16

[ ]:
```
# Potencia
3**2
```

[ ]: 9

[ ]:
```
# Resto o módulo
10%3
```

[ ]: 1

[ ]:
```
# Resto o módulo
24%6
```

[ ]: 0

[ ]:
```
# División parte entera
10//3
```

[ ]: 3

[ ]:
```
# División parte entera
4//6
```

[ ]: 0

1.1.2 Expresiones Unarias

Por otra parte, también se pueden encontrar **operadores unarios**: * **Identidad** (+) * **Negación** (-)

[ ]:
```
# Identidad
+4
```

[ ]:
```
4
```

[ ]:
```
# Identidad
+(-23)
```

[ ]:
```
-23
```

[ ]:
```
# Negación
-6
```

[ ]:
```
-6
```

[ ]:
```
# Negación
-(-5)
```

[ ]:
```
5
```

1.1.3 Precedencia de Operadores

La `precedencia de operadores` son reglas sencillas que permiten a Python "ordenar" la evaluación de expresiones. Se reconocen 3 tipos de operadores: 1. **Operadores aritméticos**: corresponden a los operadores que se han visto hasta el momento y sirven para realizar cálculos matemáticos sencillos. 2. **Operadores de comparación**: sirven para hacer evaluaciones que siempre resultan en valores de verdad. Estos serán revisados más adelante, cuando se quiera condicionar los programas. 3. **Operadores lógicos**: sirven para representar las combinaciones que pueden hacerse sobre valores de verdad y que también se verán más adelante.

Para Python, siempre se verán primero los operadores `aritméticos`, luego los de `comparación` y `finalmente los lógicos`. Además, cada categoría tiene su propio orde de evaluación preestablecido.

Python sigue las reglas de precedencia de signos, las cuales fueron vistas en la asignatura de matemáticas.

Antes de continuar, resuelva en su cuaderno la siguiente expresión matemática:

5 + 5 ** 3 / 5 * 4 - 10 * 3

[ ]:
```
# Compare el resultado obtenido con lo que arroja Python
5+5**3/5*4-10*3
```

[ ]:
```
75.0
```

Al realizar operaciones matemáticas, en el intérpete de Python, y de forma independiente los resutados pueden ser distintos lo cual puede ser por dos motivos:

- La precedencia de operadores
- El tipo de dato

En los operadores aritméticos, el orden de prioridad, desde el operador más importante al de menor importancia, se presenta en la siguiente tabla:

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binaria	Derecha	1
Identidad	+	Unaria	-	2
Negación	-	Unaria	-	2
Multiplicación	*	Binaria	Izquierda	3
División	/	Binaria	Izquierda	3
Módulo o resto	%	Binaria	Izquierda	3
División parte entera	//	Binaria	Izquierda	3
Suma	+	Binaria	Izquierda	4
Resta	-	Binaria	Izquierda	4

Las reglas de precedencia pueden ser modificadas utilizando ( ) **paréntesis**.

A modo de ejemplo, si se quisiera realizar el ejercicio anterior

5 + 5 ** 3 / 5 * 4 - 10 * 3

pero darle prioridad a la operación 5 + 5, se debe utilizar **paréntesis**

(5 + 5) ** 3 / 5 * 4 - 10 * 3

[ ]:
```
# Vuelva a desarrollar el ejercicio sin paréntesis
5 + 5 ** 3 / 5 * 4 - 10 * 3
```

[ ]:
```
75.0
```

[ ]:
```
# Desarrolle el ejercicio con paréntesis
(5 + 5) ** 3 / 5 * 4 - 10 * 3
```

[ ]:
```
770.0
```

Cuando se quiere modificar las reglas de precedecia pueden surgir los siguientes problemas:

- Los paréntesis que se abren deben cerrararse, vale decir, los paréntesis deben estar **balanceados**.
- Solo se deben utilizar paréntesis redondos ( ). El uso de `corchetes` [ ] o `paréntesis de llaves` { } se utilizan para definir tipos de datos (**listas** y **diccionarios** respectivamente)

Veamos el siguiente ejemplo

[ ]:
```
(2+((3**2)/5*3)+3*2-(3*4)+5)+5)
```

```
  File "<ipython-input-23-d360f9fd4325>", line 1
    (2+((3**2)/5*3)+3*2-(3*4)+5)+5)
                                 ^
```

```
SyntaxError: unmatched ')'
```

1.2 Tipos de datos

En todos los lenguajes de programación existen distintos tipos de datos para facilitar el trabajo a realizar. Los tipos de datos son un conjunto determinado de valores con propiedades y características determinadas.

Los 3 tipos principales de datos son:

- Tipo numérico
- Tipo texto
- Tipo lógico

Los tipos de datos texto y lógicos serán vistos con mayor detalle a medida que avance el curso, por el momento solo se dará una pequeña definición de ellos para comprender su importancia.

1.2.1 Tipo numérico

En Python existen tres clases de números:

- **Números enteros** (`int`): son una representación de los números enteros, tanto positivos como negativos.
- **Números flotantes** (`float`): son una representación de los números reales, vale decir, son números que tienen una parte entera y una parte decimal. Se debe tener en cuenta que para separar la parte entera de la decimal se utiliza el punto (.), esto se debe a que Python fue escrito por angloparlantes. A los números flotantes se les denomina números de punto flotante.
- **Números complejos** (`complex`): son una representación de conjunto matemático correspondiente a los números complejos. Lo anterior significa que el número está conformado por una parte real y otra imaginaria. Esta última es acompañada con la letta `j`

Cabe señalar que los tipos de datos numéricos son solo aproximaciones de los conjuntos matemáticos que representan debido a que la cardinalidad de estos es infinita y los computadores tienen una cantidad de memoria finita

[ ]:
```
# Operación Números Complejos
(2+5j) + (4-1j)
```

[ ]:
```
(6+4j)
```

[ ]:
```
# Operación Números Complejos
15 - (4-1j)
```

[ ]:
```
(11+1j)
```

[ ]:
```
# Operación Números Enteros y Flotantes
2 + 4 - 8 + 15.0
```

[ ]:
```
13.0
```

En el caso del último ejemplo, al desarrollar la expresión matemática entrega como resultado un número flotante. Esto se debe a que cuando se tengan operaciones que combinen distintos tipos de datos numéricos, Python **siempre mantendrá el resultado en el conjunto más general de los conjuntos numéricos**.

Por ello, dado que en el ejercicio anterior se tienen datos de tipo `entero` y `flotante`, el resultado será `flotante`.

1.2.2 Tipo texto

Son conocidos como **strings** (`str`) y se utilizan par almacenar secuencias de cadenas de texto para la generación mensajes y archivos con información

Los strings se identifican con la utilización de comillas simples (') o comillas dobles ("), las cuales delimitan el comienzo y fin de la secuencia de caracteres.

Cabe señalar que, si el string se inicia con comillas simples, debe terminar con comillas simples. Si se inicia con comillas dobles, debe terminar con comillas dobles.

[ ]:
```
# Ejemplo string
"este es un string"
```

[ ]:
```
'este es un string'
```

[ ]:
```
# Ejemplo string
'este es un string'
```

[ ]:
```
'este es un string'
```

[ ]:
```
# Ejemplo string"
'este es un mal ejempo de string"
```

```
  File "<ipython-input-29-0e5971e658f2>", line 2
    'este es un mal ejempo de string"
                                     ^
SyntaxError: EOL while scanning string literal
```

1.2.3 Tipo lógico

También denominado booleano, es un tipo de dato que permite expresas 2 valores: Verdadero (***True***) y Falso (***False***)

De manera interna, el valor **False** representa un **0** y el Valor **True** equivale a un **1**

Cabe destacar que ambos valores se deben escribir capitalizados, vale decir, su primera letra debe ser mayúscula para que Python los reconozca como valores lógicos

[ ]:
```
# Ejemplo booleano
True
```

[ ]:
```
True
```

[ ]:
```
# Ejemplo booleano
False
```

[ ]:
```
False
```

[ ]:
```
# Ejemplo booleano
true
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-32-2654a9368797> in <module>
      1 # Ejemplo booleano
----> 2 true

NameError: name 'true' is not defined
```

[ ]:
```
# Ejemplo booleano
false
```

1.3 Conversión de Tipos

Como se vio anteriormete, Python reconoce el tipo de dato con el cual se quiere trabajar y en el caso del tipo de dato numérico, se pueden mezclar para generar nuevas expresiones aritméticas, permitiendo que Python se quede con el tipo de dato más general.

No obstante, existe la posibilidad de poder indicar a Python qué tipo de dato se quiere obtener de manera directa. Para realizar dicho cometido, se utilizarán funciones nativas (más adelante se explicarán en detalle).

Las funciones de cambio de tipo (*typecast*) a utilizar son las siguientes:

- Conversión a número entero —> `int( )`
- Conversión a número flotante —> `float( )`
- Conversión a número complejo —> `complex( )`
- Conversión a texto —> `str( )`
- Conversión a booleano —> `bool( )`

1.3.1 Conversión a número entero

[ ]:
```
# Conversión número flotante a entero
int(3.6)
```

[ ]:
```
3
```

[ ]:
```
# Conversión número complejo a entero
int(5-3j)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-34-5fb4f1f8ce0f> in <module>
      1 # Conversión número complejo a entero
----> 2 int(5-3j)

TypeError: can't convert complex to int
```

[ ]:
```
# Conversión texto a número entero
int("5")
```

[ ]:
```
5
```

[ ]:
```
# Conversión booleano a entero
int(False)
```

[ ]:
```
0
```

1.3.2 Conversión a número flotante

[ ]:
```
# Conversión número entero a flotante
float(5)
```

[ ]:
```
5.0
```

[ ]:
```
# Conversión número complejo a flotante
float(5j)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-42-b19fa587fb7d> in <module>
      1 # Conversión número complejo a flotante
----> 2 float(5j)

TypeError: can't convert complex to float
```

[ ]:
```
# Conversión texto a número flotante
float("4.2")
```

[ ]:
```
4.2
```

[ ]:
```
# Conversión booleano a número flotante
float(True)
```

[ ]:
```
1.0
```

1.3.3 Conversión a número complejo

[ ]:
```
# Conversión número entero a complejo
complex(5)
```

[ ]:
```
(5+0j)
```

[ ]:
```
# Conversión número flotante a complejo
complex(2.8)
```

[ ]:
```
(2.8+0j)
```

[ ]:
```
# Conversión texto a número complejo
complex("1.5-j")
```

[ ]:
```
(1.5-1j)
```

[ ]:
```
# Conversión booleano a número complejo
complex(True,False)
```

[ ]:
```
(1+0j)
```

1.3.4 Conversión a texto

[ ]:
```
# Conversión número entero a texto
str(4)
```

[ ]:
```
'4'
```

[ ]:
```
# Conversión número flotante a texto
str(2.8)
```

[ ]:
```
'2.8'
```

[ ]:
```
# Conversión número complejo a texto
str(8.1+20j)
```

[ ]:
```
'(8.1+20j)'
```

[ ]:
```
# Conversión booleano a texto
str(True)
```

[ ]:
```
'True'
```

1.3.5 Conversión a booleano

[ ]:
```
# Conversión número entero a booleano
bool(34)
```

[ ]:
```
True
```

[ ]:
```
# Conversión número flotante a booleano
bool(9.56)
```

[ ]:
```
True
```

[ ]:
```
# Conversión número complejo a booleano
bool(0+0j)
```

[ ]:
```
False
```

[ ]:
```
# Conversión número complejo a booleano
bool(1+1j)
```

[ ]:
```
True
```

[ ]:
```
# Conversión texto a booleano
bool("False")
```

[ ]:
```
True
```

[ ]:
```
# Conversión texto a booleano
bool("True")
```

[ ]:
```
True
```

1.4 Operador asignación

Además de los operadores mencionados anteriormente, Python posee un operador especial denominado asignación (=). Dicho operador permite **almacenar** un valor o resultado de una expresión con un **nombre determinado por el/la programador/a** para posteriormente usarlo.

La estructura del operador asignación es la siguiente:

```
<identificador> = <expresión>
```

Se debe tener en consideración que el símbolo de asignación (=) NO es una igualdad o equivalencia, sino que permite guardar un valor/expresión determinada.

Conociendo el concepto del operador asignación, se pueden definir las **variables** y **constantes** para almacenar valores.

1.4.1 Variables y Expresiones

Reglas para identificadores Las variables y constantes pueden tener cualquier nombre, siempre y cuando se cumplan algunas sencillas reglas:

1. Un identificador puede estar conformado por:

- **Letras**: minúsculas y/o mayúsculas
- **Dígitos**: 0 - 9
- Caracter de subrayado o **guión bajo**

2. El **primer caracter** no puede ser un dígito
3. No puede conincidir con una palabra reservada del lenguaje Python, las cuales son: **and**, **assert**, **break**, **class**, **continue**, **def**, **del**, **elif**, **else**, **except**, **exec**, **finally**, **for**, **from**, **global**, **if**, **import**, **in**, **is**, **lambda**, **not**, **or**, **pass**, **print**, **raise**, **return**, **try**, **while**, **yield**. Las palabras reservadas de Python son aquellas que el editor IDLE marca de color naranja.
4. Debe ser representativo, ya que hace más entendible el programa. Por ejemplo, "radio", "sumando", "nombre", "edad".
5. Para el caso de las **constantes**, sus nombres se escriben siempre con mayúscula y separando las palabras con el caracter de subrayado. Representan valores que el programa **NO DEBERÍA MODIFICAR**.
6. Para el caso de las **variables**, sus nombres se escriben en minúscula y separando las palabras con el caracter de subrayado. Representan valores que **PUEDEN** cambiar o sobreescribirse en un programa.

En la siguiente tabla se presentan nombres de variables y constantes adecuados y no adecuados para programas en Python:

Variables

Ejemplo	Correcto/Incorrecto
resultado	Correcto
sumaAcumulada	Incorrecto: las palabras se deben separar por (_)
suma_parcial	Correcto
x1	Incorrecto: nombre no representativo
cosa2	Incorrecto: nombre no representativo

Constantes

Ejemplo	Correcto/Incorrecto
PI	Correcto
NUMERO_E	Correcto
gravedad	Incorrecto: no usa mayúsculas

Ejemplo	Correcto/Incorrecto
numeroPi	Incorrecto: no usa mayúsculas

Para comprender, de mejor manera, se presenta el siguiente ejemplo donde se calcula el perímetro de una cirncunferencia:

Ejemplo Variables y Constantes A continuación se muestra un ejemplo donde se pretende obtener el perímetro de una circunferencia de radio 3

[ ]:
```
#Se definen las constantes y variables a utilizar
NUMERO_PI = 3.14
radio = 3

#Se realiza los cálculos respectivos
perimetro_circunferencia = 2 * NUMERO_PI * radio

#Se entrega el resultado
perimetro_circunferencia
```

[ ]:
```
18.84
```

Del mismo modo, se presenta un nuevo ejemplo donde se desea calcular el perímetro de un rectángulo

[ ]:
```
#Se definen las variables a utilizar
lado_a = 3
lado_b = 4

#Se realizan los cálculos respectivos
perimetro_rectangulo = 2*lado_a + 2*lado_b

#Se entrega el resultado
perimetro_rectangulo
```

[ ]:
```
14
```

2 Bibliografía

2.1 Sobre la notación infija

Colaboradores de Wikipedia. (2019, 25 diciembre). *Notación de infijo*. Wikipedia, la enciclopedia libre. Recuperado 1 de agosto de 2022, de https://es.wikipedia.org/wiki/Notaci%C3%B3n_de_infijo

2.2 Sobre notación de punto flotante

Borgwardt, M. (s. f.). *Números de punto flotante*. La Guía del Punto Flotante. Recuperado 1 de agosto de 2022, de http://puntoflotante.org/formats/fp/

Colaboradores de Wikipedia. (2022, 10 mayo). *Coma flotante*. Wikipedia, la enciclopedia libre. Recuperado 1 de agosto de 2022, de https://es.wikipedia.org/wiki/Coma_flotante

2.3 Sobre tipos numéricos y operadores aritméticos

Ceder, V. L., Harms, D. K., & McDonald, K. M. (2010). The Quick Python Book. En *Built-in data types* (2.a ed., p. 19). Manning.

GeeksforGeeks. (2021, 1 octubre). *Python Data Types*. Recuperado 2 de agosto de 2022, de https://www.geeksforgeeks.org/python-data-types/

GeeksforGeeks. (2020, 30 agosto). *Python Arithmetic Operators*. Recuperado 2 de agosto de 2022, de https://www.geeksforgeeks.org/python-arithmetic-operators/

Python Software Foundation. (2022, 1 agosto). An Informal Introduction to Python. Python 3.10.5 documentation. Recuperado 1 de agosto de 2022, de https://docs.python.org/3/tutorial/introduction.html

2.4 Estándar de formato

El Libro De Python. (2022). *Python PEP8*. Recuperado 1 de agosto de 2022, de https://ellibrodepython.com/python-pep8#convenciones-al-nombrar-elementos-camelcase-y-snake_case

Van Rossum, G., Warsaw, B., & Coghlan, N. (2001, 21 julio). *PEP 8 – Style Guide for Python Code*. Python Enhancement Proposals. Recuperado 1 de agosto de 2022, de https://peps.python.org/pep-0008/