

# Índice

<b>1. Introducción a Python</b>	<b>3</b>
1.1. ¿Qué es Python? . . . . .	3
1.2. Ventajas de usar Python . . . . .	3
<b>2. Instalación y configuración del entorno de desarrollo</b>	<b>4</b>
2.1. Descarga e instalación de Python . . . . .	4
2.2. Configuración del entorno de desarrollo . . . . .	5
2.3. Tu primer programa en python . . . . .	7
2.4. Recursos de aprendizaje . . . . .	7
<b>3. Fundamentos de la sintaxis de Python</b>	<b>7</b>
3.1. Variables y asignaciones . . . . .	8
3.2. Operadores Aritmeticos . . . . .	12
3.3. Operadores de comparación . . . . .	16
3.4. Operadores lógicos . . . . .	20
3.5. Operadores de asignación . . . . .	22
3.6. Operadores de pertenencia e identidad . . . . .	25
3.7. Expresiones . . . . .	26
3.8. Comentarios y formato de código . . . . .	26
<b>4. Variables y tipos de datos</b>	<b>29</b>
4.1. Tipos numéricos . . . . .	29
4.2. Tipos de texto . . . . .	30
4.3. Tipos de secuencia . . . . .	30
4.4. Tipos de mapeo . . . . .	30
4.5. Tipos booleanos . . . . .	31
<b>5. Estructuras de control</b>	<b>31</b>
5.1. Declaraciones condicionales . . . . .	31
5.2. Bucles y iteraciones . . . . .	34
5.3. Tipos de bucles en Python . . . . .	34
<b>6. Listas, tuplas y diccionarios</b>	<b>35</b>
6.1. Listas y operaciones comunes . . . . .	35
6.2. Tuplas y sus propiedades . . . . .	40
6.3. Diccionarios y métodos de acceso . . . . .	43
<b>7. Funciones y modularidad</b>	<b>45</b>
7.1. Definición y llamada de funciones . . . . .	45
7.2. Argumentos y parámetros . . . . .	46
7.3. Módulos y su importación . . . . .	47

<b>8. Manejo de excepciones y errores</b>	<b>48</b>
8.1. Introducción a las excepciones . . . . .	48
8.2. Uso de try, except, else . . . . .	49
8.3. Tipos de excepciones y manejo específico . . . . .	50
<b>9. Entrada y salida de datos</b>	<b>52</b>
9.1. Función print() . . . . .	52
9.2. Función input() . . . . .	53
9.3. Formateo de cadenas . . . . .	54
<b>10. Trabajo con archivos</b>	<b>56</b>
10.1. Apertura, lectura y escritura de archivos . . . . .	56
10.2. Operaciones comunes de archivos . . . . .	57
10.3. Manejo de archivos con el bloque “with” . . . . .	57
<b>11. Introducción a la programación orientada a objetos</b>	<b>60</b>
11.1. Definición de clases y objetos . . . . .	60
11.2. Métodos y atributos de clase . . . . .	60
<b>12. Clases y objetos</b>	<b>60</b>
12.1. Encapsulación y visibilidad . . . . .	60
12.2. Métodos especiales . . . . .	60
12.3. Herencia y composición . . . . .	60
<b>13. Herencia y polimorfismo</b>	<b>60</b>
13.1. Herencia simple y múltiple . . . . .	60
13.2. Polimorfismo y sobrecarga de métodos . . . . .	60
13.3. Clases abstractas e interfaces . . . . .	60
<b>14. Módulos y paquetes</b>	<b>60</b>
14.1. ¿Qué son los Módulos en Python? . . . . .	60
14.2. Creación y Uso de Módulos . . . . .	60
14.3. Importación de Módulos . . . . .	60
14.4. Módulos Estándar de Python . . . . .	60
14.5. ¿Qué son los Paquetes en Python? . . . . .	60
14.6. Creación y Organización de Paquetes . . . . .	60
14.7. Importación de Paquetes . . . . .	60
<b>15. Manipulación de strings y expresiones regulares</b>	<b>60</b>
15.1. Operaciones comunes con cadenas de texto . . . . .	60
15.2. Operaciones comunes con cadenas de texto . . . . .	60
<b>16. Trabajo con fechas y tiempos</b>	<b>60</b>
16.1. Módulos datetime y time . . . . .	60
16.2. Operaciones y formateo de fechas y tiempos . . . . .	60

<b>17. Introducción a Git</b>	<b>60</b>
17.1. ¿Qué es Git y por qué es importante? . . . . .	60
17.2. Configuración inicial de Git . . . . .	60
17.3. Comandos básicos de Git . . . . .	60
<b>18. Creación de interfaces gráficas con bibliotecas como Tkinter</b>	<b>60</b>
18.1. Introducción a las interfaces gráficas . . . . .	60
18.2. Uso de Tkinter para GUIAs simples . . . . .	60

## 1. Introducción a Python

### 1.1. ¿Qué es Python?

Python es un lenguaje de programación, lo que este está diseñado para una fácil comprensión para los seres humanos. Fue creado por Guido van Rossum, lanzó su primera versión en 1991, Python ha ganado popularidad y se ha convertido en uno de los lenguajes de programación.

El diseño de Python se centra en la simplicidad y la elegancia lo que facilita a los desarrolladores escribir códigos claros y concisos.

Python es un lenguaje interpretado, lo que significa que el código escrito por los programadores se traduce a un lenguaje intermedio que luego es ejecutado por el intérprete de Python, esto permite un rápido desarrollo, ya que los programadores pueden ver el resultado de su código inmediatamente después de escribirlo.

Utiliza “indentación” para definir bloques de código, lo que elimina la necesidad de palabras claves adicionales, esto hace que el código sea limpio y fácil de leer. Python es un lenguaje multiparadigma, lo que significa que está orientada a programación con objetos como programación estructurada, esto permite a los desarrolladores utilizar diferentes enfoques según las necesidades de su proyecto. Además Python tiene una gran biblioteca estándar que proporciona módulos y paquetes para diversas aplicaciones, desde desarrollo web hasta procesamiento de datos y cálculos científicos

### 1.2. Ventajas de usar Python

1. Es un lenguaje de sintaxis amplia y legible: Al ser lenguaje de programación de alto nivel, diseñado para que los algoritmos sean expresados de forma clara y fácilmente entendibles por los seres humanos.
2. Ampliamente utilizado en múltiples campos: Su amplia variedad de usos lo ha dejado como el primero en el top 10 de los lenguajes de programación más utilizados según Tiobe, extraídos de las habilidades más desarrolladas por desarrolladores, empresas del sector y terceros.

3. Python posee una gran cantidad de bibliotecas y frameworks : La gran cantidad de usos de Python se traduce en múltiples librerías y frameworks que ayudan a llevar a cabo funcionalidades. En sí mismo ya tiene una biblioteca estándar y podemos encontrar hasta 135. 000+ más para diversas aplicaciones. Sin embargo, entre las más populares según el sitio de AWS se encuentran Matplotlib, Pandas, Request, Numpy, Keras y OpenCV-Python. Esta variedad no se limita solo a las librerías. Así mismo la podemos encontrar en los marcos o frameworks, que facilitan el proceso de creación debido a que ahorra el proceso de escritura de un código.
4. Fácil portabilidad: Python es uno de los lenguajes de programación más portátiles y versátiles disponibles. Debido a que es un lenguaje de programación interpretado, en lugar de un lenguaje compilado, se puede ejecutar en una amplia variedad de sistemas operativos y plataformas de hardware sin necesidad de realizar ajustes o cambios significativos en el código fuente.
5. Tiene una gran comunidad de desarrolladores : Es una herramienta que constantemente evoluciona para suplir las necesidades que poco a poco van surgiendo en el campo de la tecnología, como hemos visto hasta ahora en sus usos para el machine learning.
6. Multiplataforma: Python es uno de esos lenguajes de programación que puede ser ejecutado en cualquier sistema operativo en el cual se opere. Así es: no importa si se trata de Windows, Linux, macOS, y otros, este se puede ejecutar sin problema. Y, lo mejor, es que se desarrolla el código una única vez y podrá emplearse en los demás SO.

## 2. Instalación y configuración del entorno de desarrollo

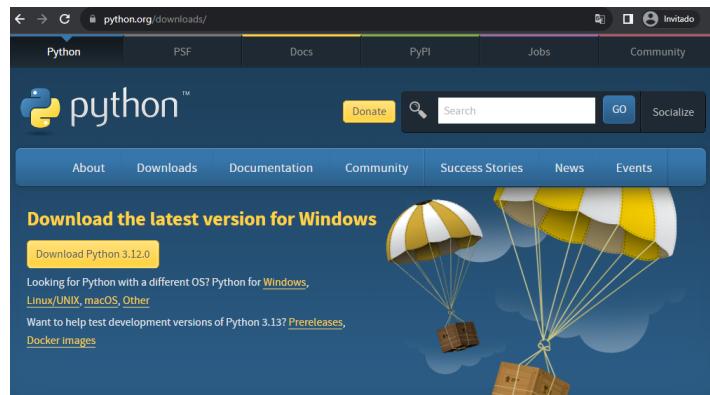
### 2.1. Descarga e instalación de Python

A continuación, te dejaré un video para aprender de manera más práctica y visual, tienes que seguir los pasos que se mencionan: Tutorial

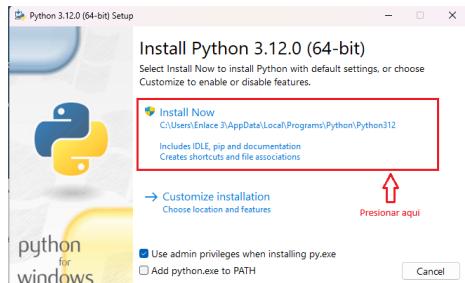


Siguiendo con algo más teórico, para comenzar a programar en Python, primero necesitas instalar Python en tu sistema. Puedes hacerlo siguiendo estos pasos:

- Descarga Python: Visita el sitio web oficial de Python en [python.org](https://python.org/downloads/) y descarga la última versión de Python. Asegúrate de elegir la versión más reciente y compatible con tu sistema operativo (Windows, macOS, o Linux).



- Instalación en Windows: Ejecuta el archivo descargado y sigue las instrucciones del instalador. Asegúrate de marcar la opción “Aregar Python al PATH” durante la instalación.



## 2.2. Configuración del entorno de desarrollo

Puedes escribir y ejecutar código Python en un entorno de desarrollo integrado (IDE) o simplemente en un editor de texto. Algunas opciones populares para IDEs incluyen:

Elige el IDE o editor de tu preferencia y configúralo.

- **IDLE:** es un programa que te ayuda a escribir y ejecutar código en Python. Es útil para principiantes porque combina un espacio para escribir código y un lugar para ver los resultados, lo que facilita aprender y probar cosas nuevas en Python. Tutorial de Cómo instalar Python y usar la herramienta IDLE: Revisa este link para más ayuda



- PyCharm: Un IDE muy popular y potente para Python. Tutorial de Como instalar y configurar Pycharm: Revisa este link para más ayuda



- Visual Studio Code (VSCode): Un editor de código que es altamente configurable y ampliamente utilizado para Python. Tutorial de como instalar y Configurar Visual Studio Code: Revisa este link para más ayuda



### **2.3. Tu primer programa en python**

Una vez que hayas instalado Python y configurado tu entorno de desarrollo, puedes comenzar a escribir código Python. Aquí tienes un ejemplo de un programa Python simple: Para ejecutar el código, guárdalo en un archivo con



extensión “.py” (por ejemplo, “mi\_programa.py”) y ejecútalo desde la línea de comandos o desde tu IDE.

### **2.4. Recursos de aprendizaje**

Python es un lenguaje versátil con una amplia comunidad y muchos recursos de aprendizaje disponibles en línea. Aquí tienes algunas fuentes para aprender Python:

- Documentación oficial de Python: [Documentación](#)
- Tutoriales en línea y cursos: Plataformas como Coursera, edX, Udemy, y Codecademy ofrecen cursos de Python.
- Libros: “Aprende Python 3 de la manera más dura” de Zed Shaw, “Automate the Boring Stuff with Python” de Al Sweigart y “Python Crash Course” de Eric Matthes son excelentes opciones.
- Foros y comunidades: Participa en la comunidad de Python en sitios como [Stack Overflow] y [Python Community] Stack Overflow Python Community
- Videos educativos en Youtube: Pildorasinformáticas, Adrián Sáenz, Holamundo, etc.

Siguiendo estos pasos, estarás listo para comenzar a programar en Python. ¡Diviértete aprendiendo y creando!

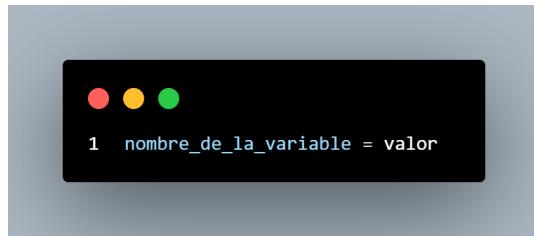
## **3. Fundamentos de la sintaxis de Python**

Python se destaca por su legibilidad y simplicidad. Algunas pautas importantes de la sintaxis incluyen:

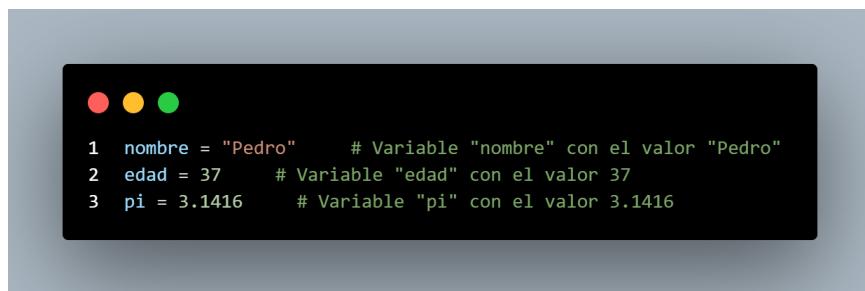
### 3.1. Variables y asignaciones

Las variables en Python son contenedores que se utilizan para almacenar datos. Estos datos pueden ser números, cadenas de texto, listas, objetos o cualquier otro tipo de información que necesites en tu programa. Las variables te permiten acceder, modificar y manipular datos de manera dinámica en tu código.

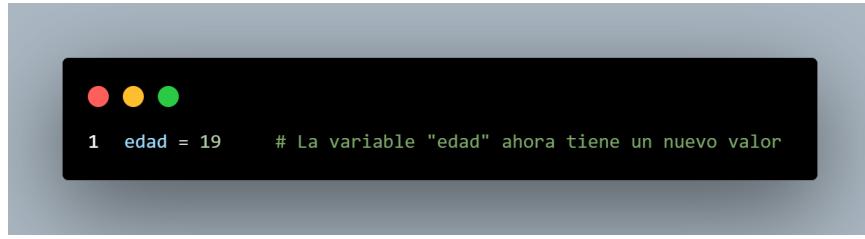
Declaración y Asignación: Para crear una variable en Python, simplemente elige un nombre descriptivo y utiliza el operador de asignación “=” para darle un valor. La estructura general es la siguiente:



- “nombre\_de\_la\_variable”: Es el nombre que eliges para la variable. Debe seguir las reglas de nomenclatura y convenciones de nombres que se mencionaron previamente.
- “valor”: Es el dato que deseas almacenar en la variable. Puede ser un número, una cadena de texto, un resultado de cálculos, una lista, un objeto, etc.



Reasignación de variables: Puedes cambiar el valor de una variable en cualquier momento asignándole un nuevo valor:



```
1 edad = 19      # La variable "edad" ahora tiene un nuevo valor
```

Identificadores: Los identificadores son nombres que se utilizan para representar variables, funciones, clases y otros elementos en Python. Algunas reglas clave para los identificadores en Python son:

- Deben comenzar con una letra (mayúscula o minúscula) o un guión bajo “\_”.
- Pueden contener letras, números y guiones bajos.
- Python distingue entre mayúsculas y minúsculas, por lo que `mi_variable` y `Mi_Variable` son consideradas diferentes.
- No se pueden utilizar palabras clave de Python como identificadores. Las palabras clave son términos reservados para funciones y estructuras de control, como “if”, “for”, “while”, entre otros.



```
1 nombre = "Pedro"
2 _variable_privada = 16
3 MiFuncion = a + b
```

Uso de variables: Las variables se utilizan para almacenar valores y realizar cálculos. Puedes utilizar variables en expresiones matemáticas, en operaciones de cadena de texto, en estructuras de control (como condicionales y bucles) y para muchas otras tareas en tu programa.

Constantes: Puedes usar variables para definir constantes, que son valores que no deben modificarse a lo largo del programa. Para indicar que una variable es una constante, es común utilizar letras mayúsculas y subrayados (por ejemplo, `PI = 3.1416`).

```
● ● ●  
1 # Usando variables en una expresión  
2  
3 a = 3  
4 b = 5  
5 suma = a + b      # La suma contiene el resultado de 5 + 3, es decir, 8  
6  
7 # Usando variables en una cadena de texto  
8 nombre = "Pedro"  
9 Mensaje = "Hola " + nombre      # Mensaje contiene "Hola Pedro"
```

```
1 x, y, z = 1, 2, 3
```

Desestructuración de Asignación: Python permite asignar valores a Múltiples variables en una sola línea de código. Esto es especialmente útil cuando se trabaja con secuencias como listas o tuplas.

Operadores de Asignación Combinada: Python ofrece operadores de asignación combinada que permiten simplificar la asignación de variables en operaciones comunes. Por ejemplo, “+=” es una forma abreviada de incrementar el valor de una variable.

```
● ● ●  
1 x = 5  
2 x += 3      # Esto es equivalente a x = x + 3
```

Tipos de datos: Python es un lenguaje de tipo dinámico, lo que significa que una variable no tiene un tipo de dato fijo. El tipo de dato de una variable se asigna automáticamente según el valor que contiene. Puedes verificar el tipo de dato de una variable utilizando la función “type()”:

```
● ● ●  
1 numero = 42  
2 cadena = "Hola"  
3 print(type(numero))      # <class 'int'> (entero)  
4 print(type(cadena))      # <class 'str'> (cadena de texto)
```

Ámbito de variables: Las variables pueden tener un ámbito local o global:

- Ámbito local: Las variables declaradas dentro de una función tienen un ámbito local y solo son accesibles dentro de esa función.

```
● ● ●  
1 def mi_funcion():  
2     variable_local = 10  
3     print(variable_local)  
4  
5 mi_funcion()      #Imprime 10  
6 print(variable_local)    # Error, variable_local no está definida en este ámbito
```

- Ámbito Global: Las variables declaradas fuera de una función o declaradas como globales dentro de una función tienen un ámbito global y son accesibles desde cualquier parte del código.

```
● ● ●  
1 variable_global = 100  
2 def modificar_variable_global():  
3     global variable_global  
4     variable_global = 200  
5  
6 modificar_variable_global()  
7 print(variable_global)      # Imprime 200
```

Ejemplo usando ambos:

```
● ● ●
1 # Variable global
2 x = 10
3
4 def mi_funcion():
5     # Variable local
6     y = 5
7     # Acceso a la variable global
8     global x
9     x = 20
10
11 mi_funcion()
12 print(x)    # Imprime 20 (el valor de la variable global ha sido modificado)
```

Estos son algunos aspectos adicionales relacionados con las variables en Python que pueden ser útiles para comprender en tu proceso de aprendizaje.

### 3.2. Operadores Aritmeticos

Los operadores aritméticos en Python se utilizan para realizar operaciones matemáticas.

Los principales son:

- Suma (+): Se utiliza para sumar dos valores.

```
● ● ●
1 # OPERADORES ARITMETICOS
2 a = 10
3 b = 3
4 suma = a + b
5 print(suma)
6 >> 13
```

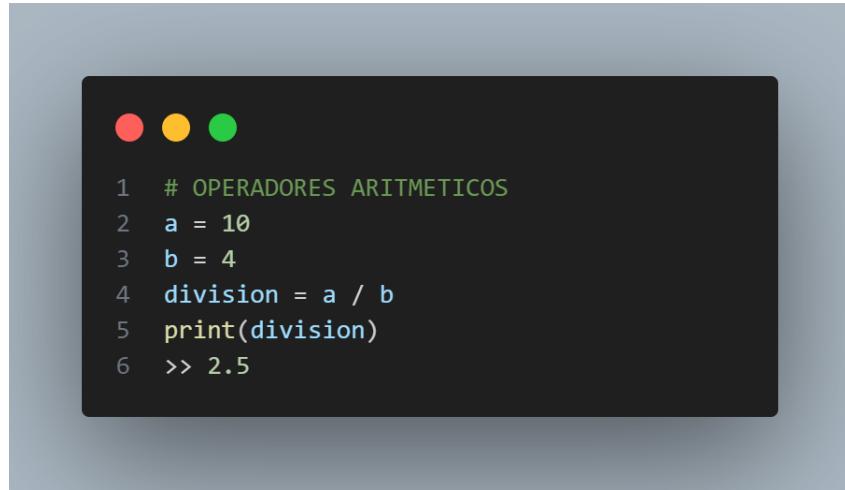
- Resta (-): Se utiliza para restar un valor de otro.

```
● ● ●  
1 # OPERADORES ARITMETICOS  
2 a = 10  
3 b = 3  
4 resta = a - b  
5 print(resta)  
6 >> 7
```

- Multiplicación (\*): Se utiliza para multiplicar dos valores.

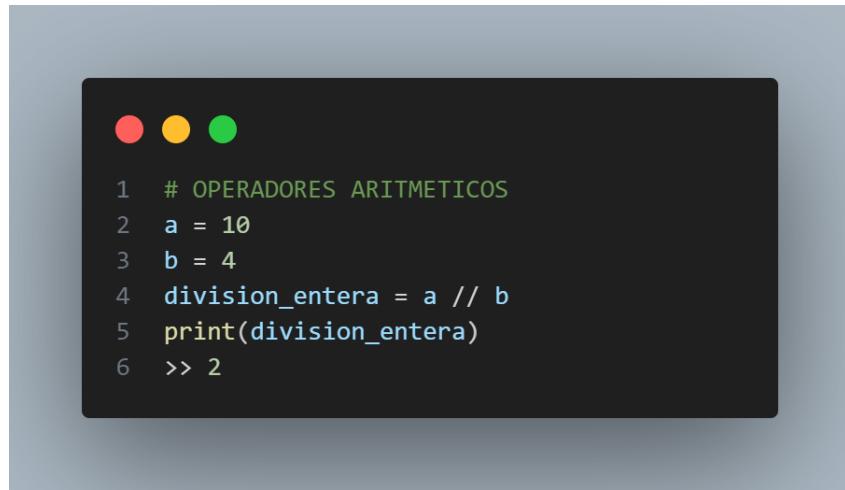
```
● ● ●  
1 # OPERADORES ARITMETICOS  
2 a = 10  
3 b = 3  
4 multiplicacion = a * b  
5 print(multiplicacion)  
6 >> 30
```

- División (/): Se utiliza para dividir un valor por otro.



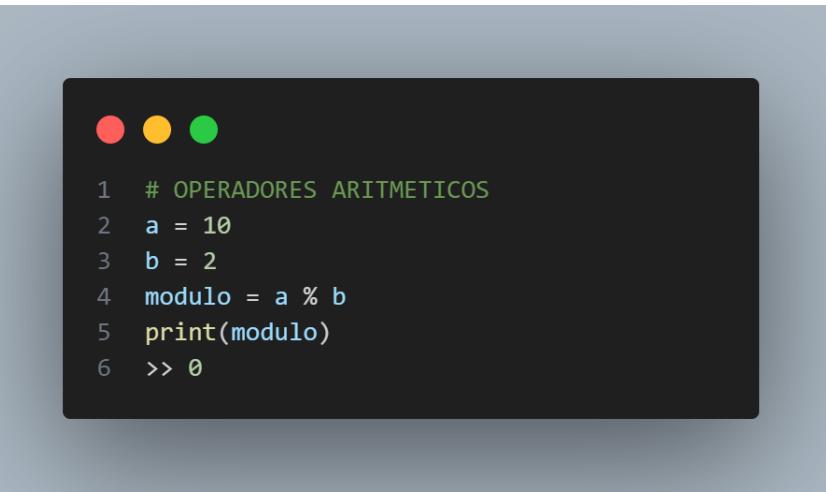
```
1 # OPERADORES ARITMETICOS
2 a = 10
3 b = 4
4 division = a / b
5 print(division)
6 >> 2.5
```

- División Entera (//): Devuelve la parte entera de la división entre dos valores.



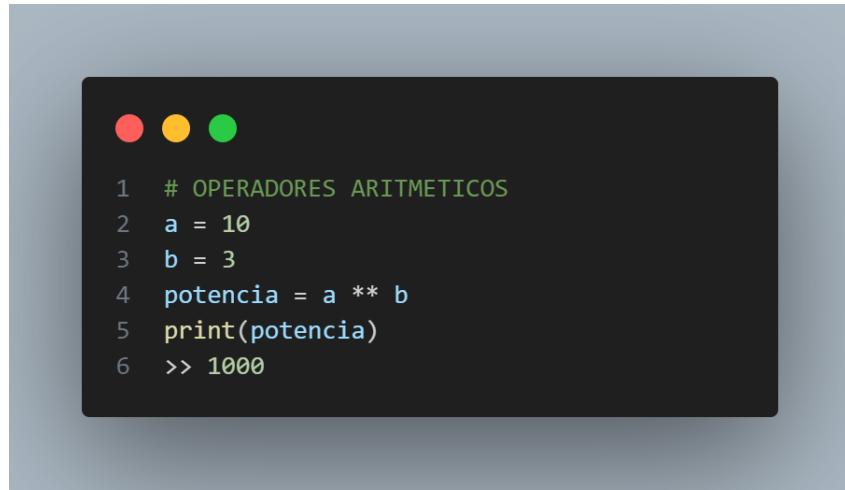
```
1 # OPERADORES ARITMETICOS
2 a = 10
3 b = 4
4 division_entera = a // b
5 print(division_entera)
6 >> 2
```

- Módulo (%): Devuelve el resto de la división entre dos valores.



```
● ● ●  
1 # OPERADORES ARITMETICOS  
2 a = 10  
3 b = 2  
4 modulo = a % b  
5 print(modulo)  
6 >> 0
```

- Potencia (\*\*): Eleva un valor a una potencia.



```
● ● ●  
1 # OPERADORES ARITMETICOS  
2 a = 10  
3 b = 3  
4 potencia = a ** b  
5 print(potencia)  
6 >> 1000
```

### 3.3. Operadores de comparación

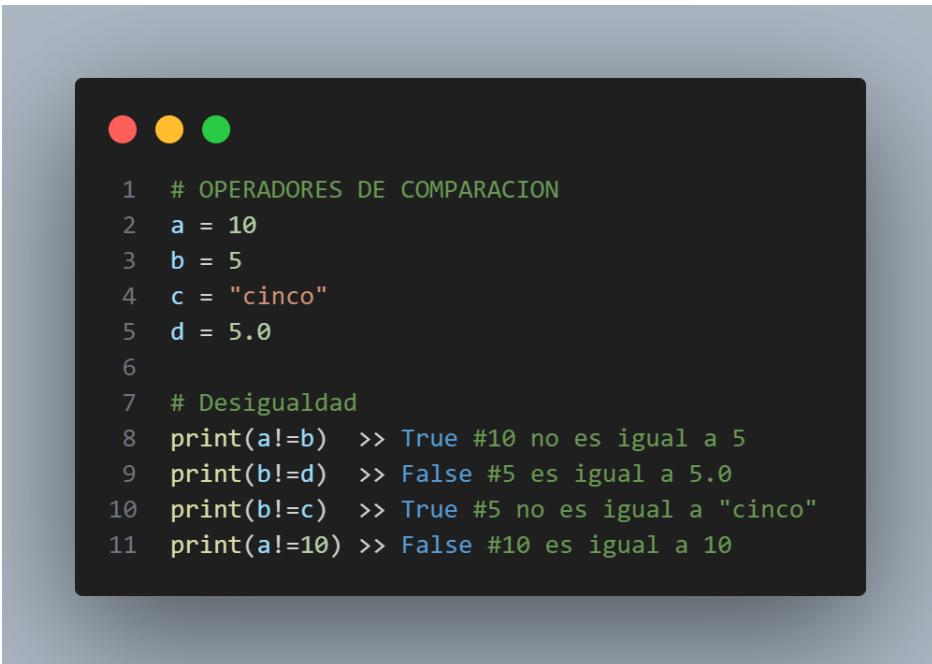
Los operadores de comparación se utilizan para comparar dos valores y devuelven un valor booleano (True o False). Los principales son:

- Igual (==): Comprueba si dos valores son iguales.



```
1 # OPERADORES DE COMPARACION
2 a = 10
3 b = 5
4 c = "cinco"
5 d = 5.0
6
7 # Igualdad
8 print(a==b)  >> False #10 no es igual a 5
9 print(b==d)  >> True #5 es igual a 5.0
10 print(b==c) >> False #5 no es igual a "cinco"
11 print(a==10) >> True #10 es igual a 10
```

- No Igual (!=): Comprueba si dos valores no son iguales.



```
1 # OPERADORES DE COMPARACION
2 a = 10
3 b = 5
4 c = "cinco"
5 d = 5.0
6
7 # Desigualdad
8 print(a!=b)  >> True #10 no es igual a 5
9 print(b!=d)  >> False #5 es igual a 5.0
10 print(b!=c) >> True #5 no es igual a "cinco"
11 print(a!=10) >> False #10 es igual a 10
```

- Mayor que (>): Comprueba si un valor es mayor que otro. En caso de strings se compara cadenas de caracteres lexicográficamente (es decir, en función del orden alfabético).

```
● ● ●
```

```
1 # OPERADORES DE COMPARACION
2 a = 10
3 b = 5
4 c = 5.0
5 d = "Manzana"
6 e = "Arroz"
7
8 # Mayor que
9 print(a>b) >> True #10 es mayor que 5
10 print(b>a) >> False #5 no es mayor que 10
11 print(b>c) >> False #5 no es mayor que 5.0
12 print(d>e) >> True #"Manzana" es mayor que "Arroz"
13 print(a>e) >> Error #No podemos comparar el valor de un string con un int o un float
```

- Menor que (<): Comprueba si un valor es menor que otro. En caso de strings se compara cadenas de caracteres lexicográficamente (es decir, en función del orden alfabético).

```
● ● ●
```

```
1 # OPERADORES DE COMPARACION
2 a = 10
3 b = 5
4 c = 5.0
5 d = "Manzana"
6 e = "Arroz"
7
8 # Menor que
9 print(a<b) >> False #10 no es menor que 5
10 print(b<a) >> True #5 es menor que 10
11 print(b<c) >> False #5 no es menor que 5.0
12 print(d<e) >> False#"Manzana" no es menor que "Arroz"
13 print(a<e) >> Error #No podemos comparar el valor de un string con un int o un float
14
```

- Mayor o igual que ( $\geq$ ): Comprueba si un valor es mayor o igual que otro. En caso de strings se compara cadenas de caracteres lexicográficamente (es decir, en función del orden alfabético).

```
● ● ●  
1 # OPERADORES DE COMPARACION  
2 a = 10  
3 b = 5  
4 c = 5.0  
5 d = "Manzana"  
6 e = "Arroz"  
7  
8 # Mayor o igual que  
9 print(a>=b) >> True #10 es mayor o igual que 5  
10 print(b>=a) >> False #5 no es mayor o igual que 10  
11 print(b>=c) >> True #5 es mayor o igual que 5.0  
12 print(d>=e) >> True #"Manzana" es mayor o igual que "Arroz"  
13 print(a>=e) >> Error #No podemos comparar el valor de un string con un int o un float
```

- Menor o igual que ( $\leq$ ): Comprueba si un valor es menor o igual que otro. En caso de strings se compara cadenas de caracteres lexicográficamente (es decir, en función del orden alfabético).

```
● ● ●  
1 # OPERADORES DE COMPARACION  
2 a = 10  
3 b = 5  
4 c = 5.0  
5 d = "Manzana"  
6 e = "Arroz"  
7  
8 # Menor o igual que  
9 print(a<=b) >> False #10 no es menor o igual que 5  
10 print(b<=a) >> True #5 es menor o igual que 10  
11 print(b<=c) >> True #5 es menor o igual que 5.0  
12 print(d<=e) >> False #"Manzana" no es menor o igual que "Arroz"  
13 print(a<=e) >> Error #no podemos comparar el valor de un string con un int o un float
```

### 3.4. Operadores lógicos

Los operadores lógicos se utilizan para combinar expresiones lógicas y devuelven un valor booleano.

Los principales son:

- and: Devuelve True si ambas expresiones son True.

```
1 # OPERADORES LOGICOS
2 a = True
3 b = False
4 c = True
5 d = False
6
7 print(a and b) >> False #Solo una variable tiene el valor True
8 print(a and c) >> True #Ambas variables tienen el valor True
9 print(b and d) >> False #Ninguna variable tiene el valor True
```

- or: Devuelve True si al menos una de las expresiones es True.

```
● ● ●  
1 # OPERADORES LOGICOS  
2 a = True  
3 b = False  
4 c = True  
5 d = False  
6  
7 print(a or b) >> True #Al menos una variable tiene el valor True  
8 print(b or d) >> False #Ninguna variable tiene el valor True
```

- not: Devuelve el inverso de la expresión.

```
● ● ●  
1 # OPERADORES LOGICOS  
2 a = True  
3 b = False  
4 c = True  
5 d = False  
6  
7 print(a or b) >> True #Al menos una variable tiene el valor True  
8 print(b or d) >> False #Ninguna variable tiene el valor True
```

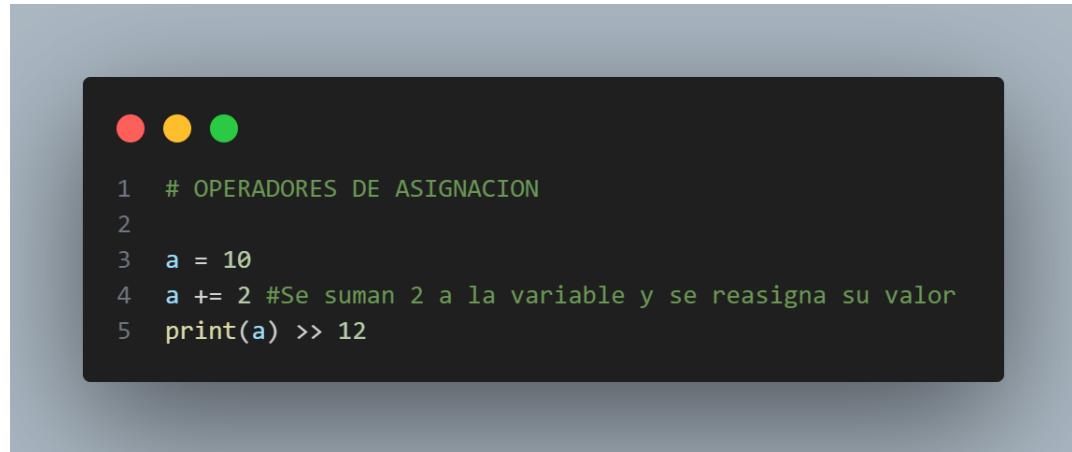
### 3.5. Operadores de asignación

Los operadores de asignación se utilizan para asignar valores a variables. Algunos de ellos incluyen:

- :=: Asigna un valor a una variable.

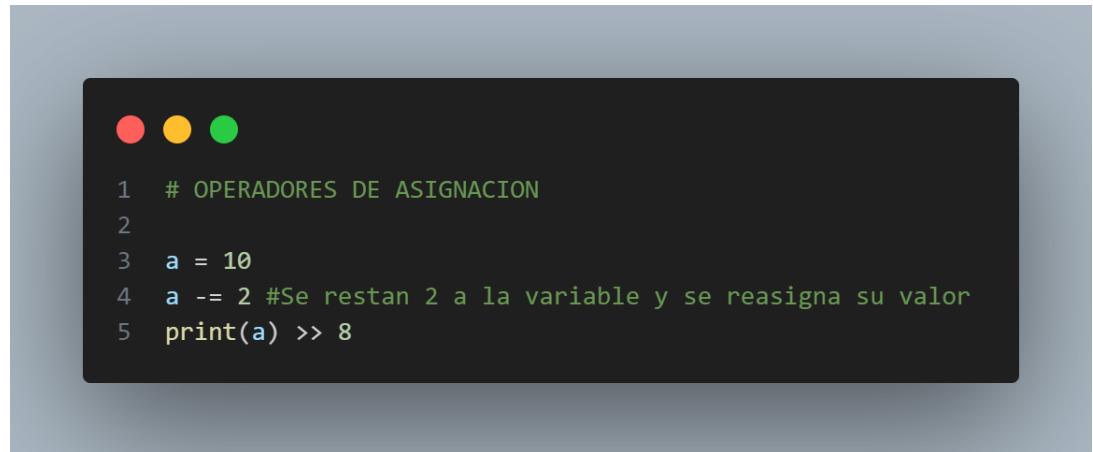
```
● ● ●  
1 # OPERADORES DE ASIGNACION  
2  
3 a = 10 # A la variable a se le asigna el valor 10
```

- `+=`: Suma y asigna el resultado a la variable.



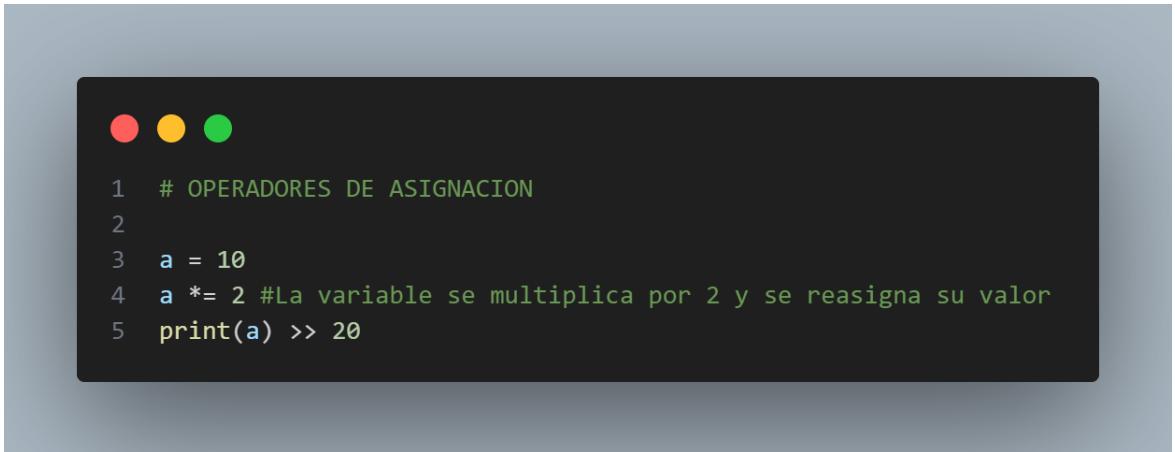
```
1 # OPERADORES DE ASIGNACION
2
3 a = 10
4 a += 2 #Se suman 2 a la variable y se reasigna su valor
5 print(a) >> 12
```

- `-=`: Resta y asigna el resultado a la variable.



```
1 # OPERADORES DE ASIGNACION
2
3 a = 10
4 a -= 2 #Se restan 2 a la variable y se reasigna su valor
5 print(a) >> 8
```

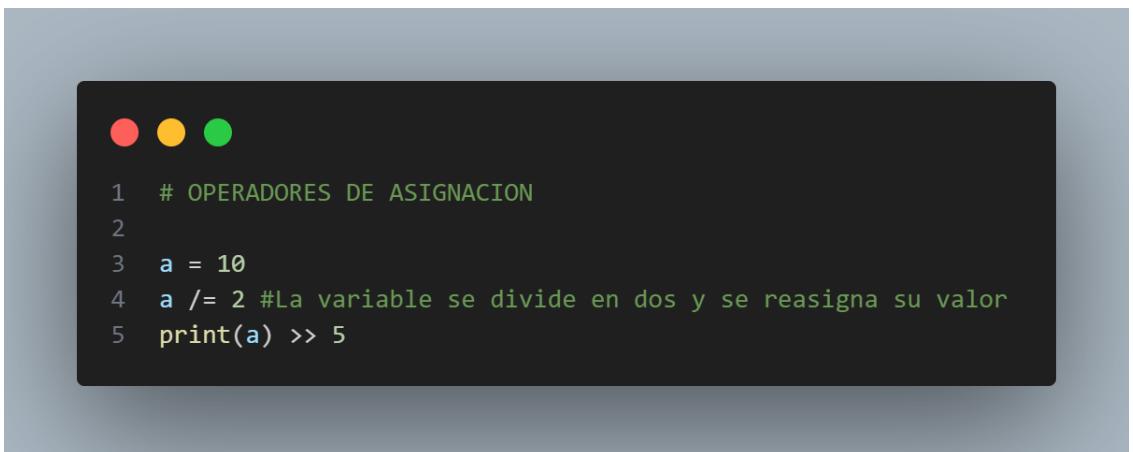
- \*=: Multiplica y asigna el resultado a la variable.



```
● ● ●

1 # OPERADORES DE ASIGNACION
2
3 a = 10
4 a *= 2 #La variable se multiplica por 2 y se reasigna su valor
5 print(a) >> 20
```

- /=: Divide y asigna el resultado a la variable.



```
● ● ●

1 # OPERADORES DE ASIGNACION
2
3 a = 10
4 a /= 2 #La variable se divide en dos y se reasigna su valor
5 print(a) >> 5
```

### 3.6. Operadores de pertenencia e identidad

- Operadores de Pertenencia: ‘in’ y ‘not in’ se utilizan para verificar si un elemento está presente en una secuencia (como una lista, tupla, etc.)

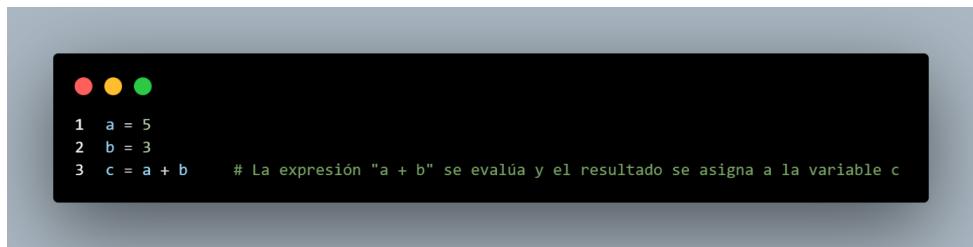
```
1 # OPERADORES DE PERTENENCIA E IDENTIDAD
2
3 lista = [1, 2, 3, 4, 5]
4
5 print(3 in lista) >> True #3 si está presente en la lista
6 print(6 in lista) >> False #6 no está presente en la lista
7
8 print(6 not in lista) >> True #6 no esta presente en la lista
9 print(3 not in lista) >> False #3 si está presente en la lista
10
```

- Operadores de Identidad: ‘is’ y ‘is not’ se utilizan para verificar si dos objetos tienen la misma identidad (es decir, si son el mismo objeto en la memoria).

```
1 # OPERADORES DE PERTENENCIA E IDENTIDAD
2
3 a = [1, 2, 3]
4 b = [1, 2, 3]
5
6 print(a is b) >> False #a y b no son el mismo objeto en la memoria
7 print(a is a) >> True #corresponde al mismo objeto en la memoria
8
9 print(a is not b) >> True #a y b no son el mismo objeto en la memoria
10 print(a is not a) >> False #corresponde al mismo objeto en la memoria
```

### 3.7. Expresiones

Una expresión es una combinación de valores, variables y operadores que se evalúa para producir un resultado. Las expresiones se utilizan en muchas partes del código, como asignaciones, condicionales, bucles y funciones. Por ejemplo:



```
● ● ●
1 a = 5
2 b = 3
3 c = a + b      # La expresión "a + b" se evalúa y el resultado se asigna a la variable c
```

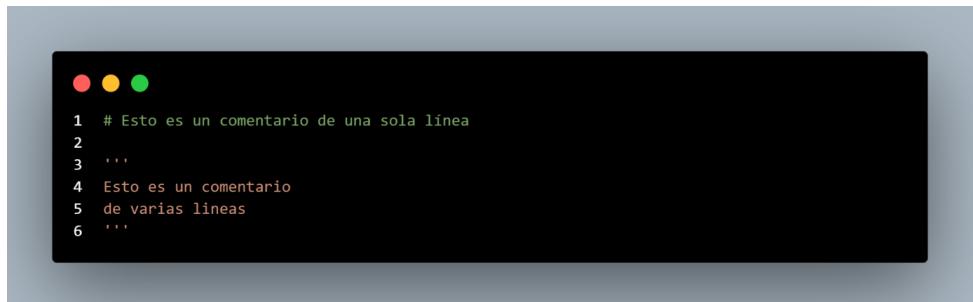
Las expresiones pueden ser tan simples como una sola variable o tan complejas como cálculos matemáticos o lógicos.

Estos son los operadores y las expresiones en Python que son fundamentales para realizar tareas de cálculo, toma de decisiones y procesamiento de datos en tus programas. Puedes combinar estos operadores y expresiones de diversas maneras para realizar tareas específicas en tu código.

### 3.8. Comentarios y formato de código

Los comentarios y el formato de código son aspectos cruciales de la programación en Python. Los comentarios son notas en el código que se utilizan para explicar lo que hace el código, mientras que el formato se refiere a la estructura y el estilo del código para que sea más legible y mantenable.

Comentarios: Los comentarios en Python se utilizan para agregar explicaciones o notas al código fuente. Seinizcan con el símbolo “#” y Python los ignoran durante la ejecución. Los comentarios son útiles para documentar el código y facilitar su comprensión tanto para el programador como para otros colaboradores.



```
● ● ●
1 # Esto es un comentario de una sola línea
2 ...
3 ...
4 # Esto es un comentario
5 de varias líneas
6 ...
```

- Comentarios de una sola línea: Se utilizan para incluir explicaciones breves en una línea.
- Comentarios de varias líneas: Se utilizan para describir secciones de código más largas y pueden abarcar varias líneas.

**Formato de Código:** El formato del código es importante para mantener la legibilidad y la consistencia en un proyecto. Python se destaca por su énfasis en la indentación (espacios al principio de una línea) para delimitar bloques de código.

- Indentación: La sangría en Python es fundamental para definir bloques de código. A diferencia de otros lenguajes que utilizan llaves o paréntesis para delimitar bloques, en Python, la sangría se utiliza para indicar la estructura del programa. Esto fomenta la legibilidad y la claridad del código. Un bloque de código se define mediante el uso de espacios o tabulaciones con un mismo nivel de sangría.

```

1 if condicion:
2     # Este bloque está indentado
3     print("hola")

```

- Interpretado: Python es un lenguaje de programación interpretado. Esto significa que no es necesario compilar el código fuente en código de máquina antes de ejecutarlo. En cambio, el intérprete de Python lee y ejecuta el código línea por línea en tiempo real. Esto facilita la depuración y la portabilidad, ya que un programa escrito en Python se puede ejecutar en cualquier sistema que tenga un intérprete de Python instalado.

- Tipado dinámico: Python es un lenguaje de tipado dinámico, lo que significa que no necesitas declarar explícitamente el tipo de una variable al crearla. El tipo de una variable se determina automáticamente en el tiempo de ejecución según el valor que contiene. Esto permite una mayor flexibilidad, pero también significa que debes tener cuidado con los tipos de datos para evitar errores inesperados.



```
1 variable = 42      # La variable es de tipo int
2 variable = "Hola"    # La variable ahora es de tipo str
```

- Convenciones de Nombres: En Python, se siguen ciertas convenciones para nombrar variables, funciones y clases. Algunas pautas comunes incluyen:
  - Los operadores y las expresiones son componentes esenciales en Python y en la programación en general. Los operadores se utilizan para realizar operaciones en variables y valores, mientras que las expresiones son combinaciones de variables, valores y operadores que se evalúan para producir un resultado.
  - Variables y funciones: Usar letras minúsculas y guiones bajos para separar palabras (ejemplo: mi\_variable, mi\_funcion).
  - Clases: Usar CamelCase (iniciar cada palabra con mayúscula, sin espacios ni guiones bajos) (ejemplo: MiClase).
  - Constantes: Los nombres de las constantes deben estar en mayúsculas. Si es necesario separar palabras en el nombre de la constante, se pueden usar guiones bajos. Por ejemplo, PI o TASA\_DE\_INTERES.
  - Convenciones específicas: Para indicar que una variable o función es “privada” (es decir, no debería accederse desde fuera de su módulo), se puede anteponer un guion bajo al nombre, por ejemplo, \_variable\_privada. Para variables que actúan como constantes y no deberían modificarse, se puede utilizar un guion bajo en mayúsculas al principio del nombre, como \_CONSTANTE.
  - Variables temporales: En bucles y expresiones cortas, es común usar nombres de variables temporales cortos como i, j, k para iteradores. Sin embargo, es importante que estos nombres sean descriptivos dentro del contexto.
- Espaciado: Agregar espacios alrededor de operadores y después de comas para mejorar la legibilidad.



```
1 variable = 42
2 lista = [1, 2, 3]
3 resultado = x + y
```

- Longitud de Línea: Es una buena práctica limitar la longitud de una línea de código a aproximadamente 79-80 caracteres para facilitar la lectura. Puedes usar una barra invertida \ para dividir una línea larga en varias líneas:



```
1 texto_largo = "Esta es una línea de texto muy larga que se divide en varias líneas \n para mejorar la legibilidad."
```

- Comentarios Descriptivos: Los comentarios deben ser informativos y explicar el propósito de una sección de código. Es especialmente útil para documentar partes complicadas o algoritmos. No es necesario comentar lo obvio.

El formato de código y los comentarios son esenciales para que tu código sea comprensible, colaborativo y mantenible. Adherirse a las convenciones de estilo y escribir comentarios informativos ayuda a otros programadores (y a ti mismo) a entender y mantener el código de manera más eficaz.

## 4. Variables y tipos de datos

Es necesario saber diferenciar las variables y tipos de datos al momento de programar

### 4.1. Tipos numéricos

Los tipos de datos numéricos son usados en programación para hacer cálculos y manipular números, sus tipos son:

- Enteros(int): estos representan los números enteros no decimales por ejemplo: -1, 0, 69, 13, -7

- Punto Flotante (float): Representan números con decimales. Ejemplo: 3.14, -0.5, 2.718, 100.0.

Para ejecutar estos tipos de datos numéricos basta con asignarle un valor.

Ejemplo:

#### 4.2. Tipos de texto

Los tipos de variables de texto son fundamentales en programación para manejar y manipular cadenas de caracteres.

En las variables de texto tenemos el string o “texto de verdad”, el cual puede contener desde una palabra a un párrafo completo, ademas en una variable de tipo string podemos cargar una cantidad variable de caracteres. Estos caracteres una vez cargados a un string no se pueden cambiar

Ejemplo:

#### 4.3. Tipos de secuencia

Las secuencias son estructuras de datos fundamentales que permiten almacenar colecciones ordenadas de elementos.

Una son las listas las cuales son secuencias mutables de elementos. Los elementos pueden ser de diferentes tipos, incluso otras listas.

Ejemplo:

- Agregar Elementos: lista.append(elemento)
- Acceso por Índice: lista[indice]
- Slicing: lista[inicio:fin]
- Modificar Elementos: lista[indice] = nuevo\_valor
- Eliminar Elementos: del lista[indice]

La otra secuencia son las tuplas las cuales son secuencias de elementos ordenadas e inmutables y que se utilizan para datos que por su naturaleza, no deben cambiar.

Una tupla puede ser creada poniendo los valores separados por comas y entre paréntesis. Por ejemplo, podemos crear una tupla que tenga el nombre y el apellido de una persona de la siguiente manera:

#### 4.4. Tipos de mapeo

Uno de los tipos de mapeo es el Diccionario, a veces llamado Mapa en algunos lenguajes el cual es:

Una estructura para programación general que contiene un número dinámico de entradas, donde cada entrada tiene una clave única y un valor asociado. Se

puede agregar una nueva entrada, eliminar y cambiar su valor. Y usualmente son escritos entre corchetes.

Ejemplo:

#### 4.5. Tipos booleanos

Un operador de tipo booleano es un dato que solo puede tener dos valores ya que representa valores de lógica binaria, y por lo general se pueden mostrar con un dato que sea Verdadero o Falso.

### 5. Estructuras de control

#### 5.1. Declaraciones condicionales

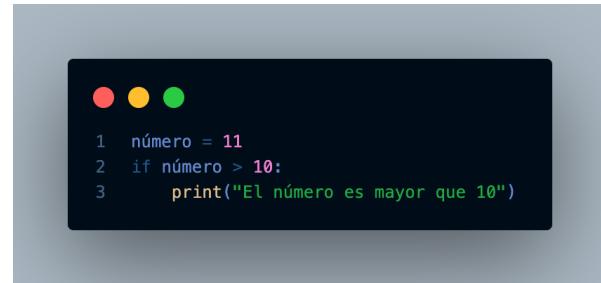
Las sentencias condicionales en Python permiten controlar el flujo del programa en función del valor de una expresión booleana. Una expresión booleana es una expresión que puede evaluarse como verdadera o falsa.

Sentencia if: se considera la más sencilla de las tres y toma decisiones basadas en si la condición es verdadera o falsa. Si la condición es verdadera, se ejecuta el bloque de código sangrado. Si la condición es falsa, se omite la ejecución del bloque.

La sintaxis básica de la condición if es la siguiente:



Por ejemplo, el siguiente código imprime “El número es mayor que 10” si el número es mayor que 10:



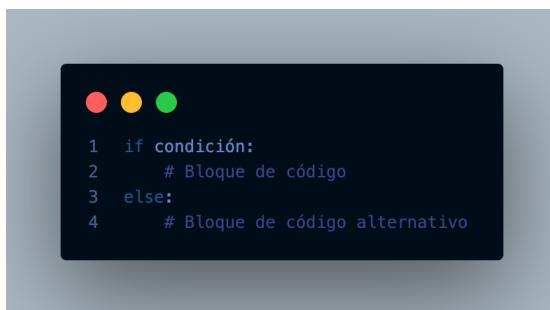
A Scratch script with three colored circles at the top. The script consists of one `if` block with the condition `número > 10`. Inside the block is a `print` command with the message "El número es mayor que 10".

```
1  número = 11
2  if número > 10:
3      print("El número es mayor que 10")
```

En este ejemplo, la condición es que el número sea mayor que 10. El bloque de código contiene una sola instrucción, que es imprimir el mensaje “El número es mayor que 10”.

Sentencia else (opcional): Puedes agregar un bloque de código que se ejecutará si la primera condición es falsa.

La sintaxis básica de la condición else es la siguiente:



A Scratch script with three colored circles at the top. It features an `if` block with the condition `condición`. The `if` block contains a comment `# Bloque de código`. Below it is an `else` block containing a comment `# Bloque de código alternativo`.

```
1  if condición:
2      # Bloque de código
3  else:
4      # Bloque de código alternativo
```

Por ejemplo, el siguiente código imprime “El número es menor que 10” si el número es menor que 10:



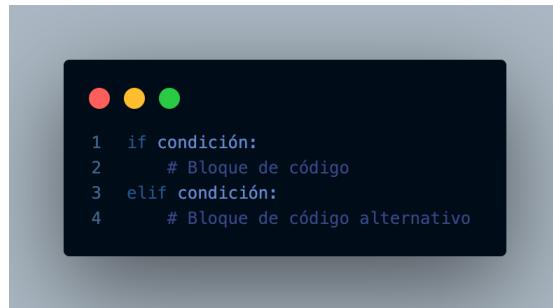
A Scratch script with three colored circles at the top. It features an `if` block with the condition `número > 10`. Inside the `if` block is a `print` command with the message "El número es mayor que 10". Below the `if` block is an `else` block containing a `print` command with the message "El número es menor que 10".

```
1  número = 5
2
3  if número > 10:
4      print("El número es mayor que 10")
5  else:
6      print("El número es menor que 10")
```

En este ejemplo, la condición es que el número sea mayor que 10. El bloque de código principal se ejecuta si la condición es verdadera. Si la condición es falsa, se ejecuta el bloque de código alternativo.

Sentencia elif (opcional): Para verificar múltiples condiciones, puedes utilizar elif después de la declaración if. Ten en cuenta que solo se ejecutará el bloque de código correspondiente a la primera condición verdadera.

La sintaxis básica de la sentencia elif es la siguiente:



```
1 if condición:
2     # Bloque de código
3 elif condición:
4     # Bloque de código alternativo
```

Por ejemplo, el siguiente código imprime “El número es mayor que 10” si el número es mayor que 10, “El número es igual a 10” si el número es igual a 10, y “El número es menor que 10” si el número es menor que 10:



```
1 número = 10
2
3 if número > 10:
4     print("El número es mayor que 10")
5 elif número == 10:
6     print("El número es igual a 10")
7 else:
8     print("El número es menor que 10")
```

En este ejemplo, la condición principal es que el número sea mayor que 10. Si la condición principal es verdadera, se ejecuta el bloque de código principal. Si la condición principal es falsa, se evalúa la primera condición elif. Si la primera condición elif es verdadera, se ejecuta el bloque de código alternativo. Si la primera condición elif es falsa, se evalúa la segunda condición elif, y así sucesivamente. Si ninguna de las condiciones elif es verdadera, se ejecuta el bloque de código else.

## 5.2. Bucles y iteraciones

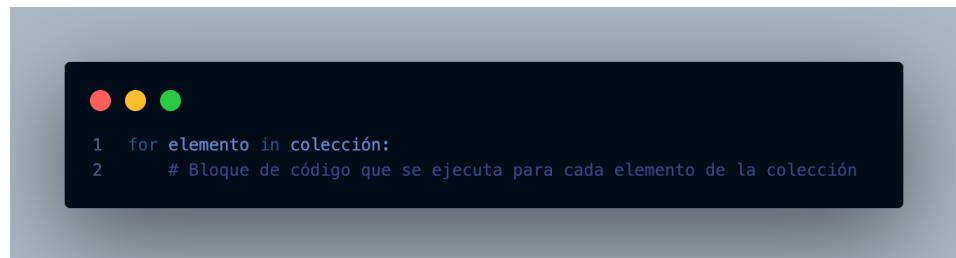
En el contexto de la programación, los términos loops e iteraciones a menudo se utilizan indistintamente, pero hay una distinción sutil:

- Iteración: La iteración se refiere al proceso de repetir una acción o un bloque de código para cada elemento de una secuencia o hasta que se cumpla una condición. La iteración no está limitada a los bucles, ya que puede ocurrir en otros contextos, como al recorrer elementos en una lista o realizar operaciones en un conjunto de datos.
- Loop (bucle): Un loop es una estructura de control que permite repetir un bloque de código múltiples veces. Puede ser un ciclo for o “while”, y generalmente se utiliza cuando se conoce la cantidad de repeticiones necesarias. Un loop puede realizar iteraciones.

## 5.3. Tipos de bucles en Python

En Python, existen dos tipos de bucles:

- Bucle for: El bucle for se utiliza para iterar sobre una colección de datos. La sintaxis es la siguiente:



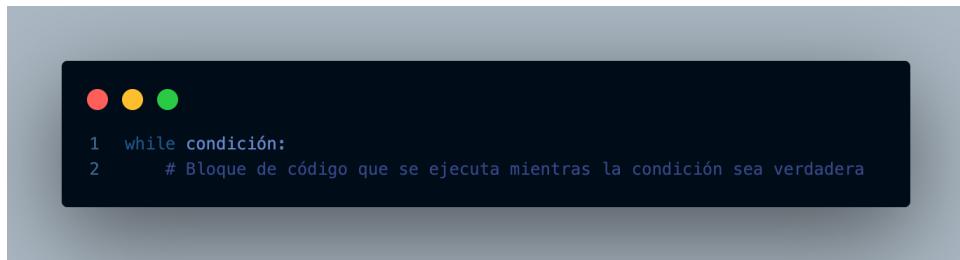
```
1 for elemento in colección:  
2     # Bloque de código que se ejecuta para cada elemento de la colección
```

Por ejemplo, el siguiente código imprime los números del 1 al 10:



```
1 for número in range(1, 11):  
2     print(número)
```

- Bucle while: El bucle while se utiliza para iterar mientras una condición sea verdadera. La sintaxis es la siguiente:



```
● ● ●
1 while condición:
2     # Bloque de código que se ejecuta mientras la condición sea verdadera
```

Por ejemplo, el siguiente código imprime los números del 1 al 10:



```
● ● ●
1 número = 1
2 while número <= 10:
3     print(número)
4     número += 1
```

## 6. Listas, tuplas y diccionarios

### 6.1. Listas y operaciones comunes

Las listas en python son un conjunto ordenado y editable de elementos, son dinámicas esto quiere decir que puede contener diferentes tipos de datos, y aparte cabe recalcar que puede tener elementos duplicados y no generará error. Estas se pueden ocupar tanto como para recopilar datos como para organizar y algunas de sus funciones son las siguientes:

- Creación de listas: Se puede crear una lista abriendo corchetes “[]” y separando los datos por comas “,”



```
1 mi_lista = ["a", 1, "b", 2]
```

- Acceso a elementos: se puede acceder a un elemento en específico de la lista poniendo el nombre de este y dentro de corchetes el número en el lugar que se encuentra (se empieza a contar desde el 0, y para seleccionar el último será -1).



```
1 mi_lista = ["a", 1, "b", 2]
2 primer_elemento = mi_lista[0]
3 #primer_elemento = "a"
4 ultimo_elemento = mi_lista[-1]
5 #ultimo_elemento = 2
```

- Modificar elementos: Seleccionando el puesto en el que se encuentra el dato e igualarlo al valor que queramos.



```
1 mi_lista = ["a", 1, "b", 2]
2 mi_lista[3] = 35
3 #mi_lista = ["a", 1, "b", 35]
4
```

- Añadir elementos: Utilizando el método “.append(elemento)” se pueden agregar datos.

```
● ● ●  
1 mi_lista = ["a", 1, "b", 2]  
2 mi_lista.append("c")  
3 #mi_lista = ["a", 1, "b", 2, "c"]
```

- Insertar elementos: Si bien cumple la misma función que se describió anteriormente se ocupa el método “.insert(posición, elemento)” con esta también se puede detallar la posición en que queremos que esté.

```
● ● ●  
1 mi_lista = ["a", 1, "b", 2]  
2 mi_lista.insert(3, "x")  
3 #mi_lista = ["a", 1, "b", "x", 2]
```

- Eliminar elementos: Para eliminar elementos hay dos métodos el “.pop(lugar del elemento)” que elimina el dato que haya en ese puesto y el “.remove(elemento)” que elimina el elemento que se mencionó en los paréntesis.

```
● ● ●  
1 mi_lista = ["a", 1, "b", 2]  
2 mi_lista.pop(2)  
3 #mi_lista = ["a", 1, 2]
```

```
● ● ●  
1 mi_lista = ["a", 1, "b", 2]  
2 mi_lista.remove(1)  
3 #mi_lista = ["a", "b", 2]
```

- Largo de una lista: Para saber cuántos elementos hay en la lista ocupamos el método “.len(mi\_lista)”.

```
● ● ●  
1 mi_lista = ["a", 1, "b", 2]  
2 largo_lista = len(mi_lista)  
3 #largo_lista = 4
```

- Rebanado(slicing): Puedes acceder a una porción de una lista utilizando el operador de rebanado “[:]”.

```
● ● ●  
1 mi_lista = ["a", 1, "b", 2]  
2 mi_lista[1:4]  
3 #[1, "b", 2]
```

- Concatenación de listas: Se pueden unir dos o más listas utilizando el operador de concatenación “+”.

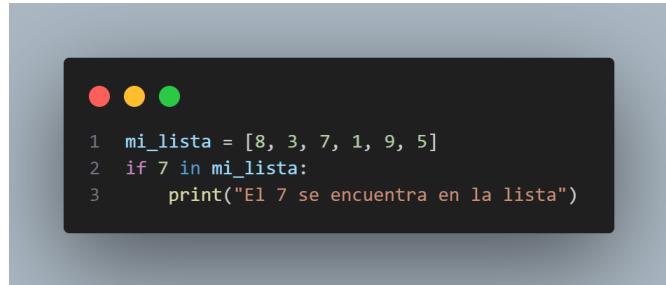
```
● ● ●  
1 lista1 = [1, 2, 3]  
2 lista2 = [4, 5, 6]  
3 mi_lista = lista1 + lista2  
4 #mi_lista = [1, 2, 3, 4, 5, 6]
```

- Ordenar una lista: Para ordenar una lista de manera ascendente o descendente se pueden utilizar los métodos “.sort()” o “.sorted()” respectivamente.

```
● ● ●  
1 mi_lista = [8, 3, 7, 1, 9, 5]  
2 mi_lista.sort()  
3 #mi_lista = [1, 3, 5, 7, 8, 9]
```

```
● ● ●  
1 mi_lista = [8, 3, 7, 1, 9, 5]  
2 mi_lista_descendente = sorted(mi_lista, reverse=True)  
3 #mi_lista_descendente = [9, 8, 7, 5, 3, 1]
```

- Buscar elementos: Se puede buscar un elemento de la lista utilizando el operador “in” o el método “.index(elemento)”.



```
● ● ●
1 mi_lista = [8, 3, 7, 1, 9, 5]
2 if 7 in mi_lista:
3     print("El 7 se encuentra en la lista")
```



```
● ● ●
1 mi_lista = [8, 3, 7, 1, 9, 5]
2 mi_lista.index(7)
3 #2
```

## 6.2. Tuplas y sus propiedades

Si bien las tuplas son un conjunto de datos similares a las listas estas son inmutables esto quiere decir que una vez creada no se puede volver a modificar, algunas de sus propiedades son:

- Creación de tuplas: Primero que nada para crear una tupla solo se necesita de paréntesis “()” y separar los datos por comas “,”, aunque también se pueden crear solo separando los elementos con comas.



```
● ● ●
1 mi_tupla = (1, "a", 2, "b")
2 mi_tupla = 1, "a", 2, "b"
```

- Acceso a elementos: Al igual que en las listas para acceder a un elemento en específico solo se necesita de corchetes “[índice]”.



```
1 mi_tupla = (1, "a", 2, "b")
2 mi_tupla[1]
3 # "a"
```

- Datos inmutables: Como dije anteriormente los datos no se pueden cambiar por lo que si seleccionamos un elemento y lo tratamos de igualar a otro nos tirara un error.



```
1 mi_tupla = (1, "a", 2, "b")
2 mi_tupla[1] = "x"
3 #Genera un error
```

- Largo de una tupla: Para saber el largo de la tupla tenemos que utilizar la función “len(tupla)”.



```
1 mi_tupla = (1, "a", 2, "b")
2 largo_tupla = len(mi_tupla)
3 #largo_tupla = 4
```

- Rebanado (slicing): Esta propiedad sirve para seleccionar una porción de la tupla creando una nueva con estos datos y para esto necesitamos utilizar el operador de rebanado “[:].”

```
● ● ●  
1 mi_tupla = (1, "a", 2, "b")  
2 sub_tupla = mi_tupla[1:5]  
3 #sub_tupla = ['a', 2, 'b', 3]
```

- Concatenación de tuplas: Del mismo modo que en las listas para concatenar las tuplas se necesita del operador de concatenación “+”.

```
● ● ●  
1 tupla1 = (1, 2, 3)  
2 tupla2 = (4, 5, 6)  
3 mi_tupla = tupla1 + tupla2  
4 #mi_tupla = (1, 2, 3, 4, 5, 6)
```

- Asignación de múltiple: Este quiere decir que en una sola línea de código se puede asignar a distintas variables los elementos de la tupla.

```
● ● ●  
1 mi_tupla = (5, 10, 15)  
2 a, b, c = mi_tupla  
3 # a = 5 b = 10 c = 15
```

### 6.3. Diccionarios y métodos de acceso

Los diccionarios en vez de las tuplas y listas contienen pares de datos que se identifican como clave-valor cada clave dentro del diccionario tiene un único valor, esto sirve para organizar todo de mejor manera y acceder a los valores de una forma mas rápida como por ejemplo:

- Creación de diccionarios: Primero que nada para crear un diccionario necesitamos utilizar llaves “{}” y dentro separando clave-valor por “:” y cada par por “,”.

```
● ● ●  
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}
```

- Acceso al valor: Esta vez al igual que en las anteriores se ocupan corchetes “[clave]” pero en vez de poner un índice esta vez se pone la clave para obtener su respectivo valor.

```
● ● ●  
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}  
2 mi_nombre = mi_diccionario["Nombre"]  
3 #mi_nombre = "Maria Jose"
```

- Modificación del valor: Para modificar un valor solo necesitamos seleccionar su clave e igualarlo al nuevo valor que queremos que obtenga.

```
● ● ●  
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}  
2 mi_diccionario["Signo"] = "Cancer"  
3 #mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Cancer"}
```

- Agregar pares: Si queremos agregar nuevos pares de clave-valor solo necesitamos seleccionar una nueva clave del diccionario e igualarlo a un nuevo valor.

```
● ● ●  
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}  
2 mi_diccionario["Ciudad"] = "Las Garzas"  
3 #mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio", "Ciudad": "Las Garzas"}
```

- Eliminación de pares: Cuando queramos eliminar un par que este en el diccionario solo debemos utilizar la declaración “del” y especificar la clave.

```
● ● ●  
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}  
2 del(mi_diccionario["Edad"])  
3 #mi_diccionario = {"Nombre": "Maria Jose", "Signo": "Capricornio"}
```

- Verificación de existencia de clave: Para verificar si una clave en específico existe dentro de un diccionario podemos utilizar el operador “in”.

```
● ● ●  
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}  
2 if "Nombre" in mi_diccionario:  
3     print("El par clave-valor si se encuentra en la lista")
```

- Obtener clave-valor: Para obtener o todas las claves o todos los valores de un diccionario podemos utilizar los métodos “.keys()” o “.values()” respectivamente, pero si queremos obtener el par de elementos utilizaremos el método “.items()”.

```
● ● ●  
1 mi_diccionario = {"Nombre": "Maria Jose", "Edad": 17, "Signo": "Capricornio"}  
2 claves = mi_diccionario.keys()  
3 #claves = ['Nombre', 'Edad', 'Signo']  
4 valores = mi_diccionario.values()  
5 #valores = ['Maria Jose', 17, 'Capricornio']  
6 pares = mi_diccionario.items()  
7 #pares = [('Nombre', 'Maria Jose'), ('Edad', 17), ('Signo', 'Capricornio')]
```

## 7. Funciones y modularidad

Las funciones son bloques de código reutilizables que realizan una tarea específica. Permiten dividir un programa en partes más pequeñas y manejables, lo que facilita la comprensión y el mantenimiento del código. La modularidad se refiere a la práctica de dividir un programa en módulos o funciones independientes que pueden ser desarrolladas y probadas de forma separada. Esto promueve el código limpio y organizado, facilitando la colaboración en equipos de desarrollo.

### 7.1. Definición y llamada de funciones

Definición de Funciones:

En la mayoría de los lenguajes de programación, las funciones se definen con la palabra clave “def” (en Python), “function” (en JavaScript), “fun” (en Kotlin), o “void” (en C++), seguido del nombre de la función y una lista de parámetros entre paréntesis. Por ejemplo, en Python:

```
● ● ●  
1 def saludar(nombre):  
2     print("Hola, " + nombre + "!")
```

Llamada de Funciones:

Para utilizar una función, se realiza una llamada a la función, pasando los valores necesarios como argumentos. Los argumentos son los valores reales que se pasan a la función durante la llamada. Por ejemplo:



En este caso, “Juan” es el argumento que se pasa a la función “saludar”.

## 7.2. Argumentos y parámetros

Parámetros de Funciones:

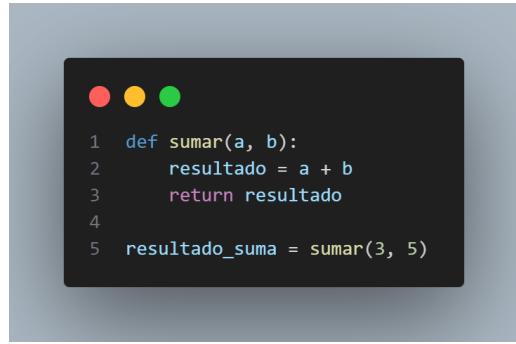
Los parámetros son variables que se utilizan en la definición de la función para aceptar valores. En el ejemplo anterior, “nombre” es un parámetro de la función “saludar”.



Argumentos de Funciones:

Los argumentos son los valores reales que se pasan a la función durante su llamada. En el ejemplo de la llamada a la función ‘saludar("Juan")’, "Juan" es el argumento que se pasa al parámetro “nombre” de la función “saludar”

Las funciones pueden tener múltiples parámetros y se pueden pasar argumentos de diferentes tipos (números, cadenas, listas, etc.). Por ejemplo:



```
● ● ●
1 def sumar(a, b):
2     resultado = a + b
3     return resultado
4
5 resultado_suma = sumar(3, 5)
```

En este caso, la función “sumar” tiene dos parámetros “a” y “b”, y se llama con los argumentos 3 y 5, respectivamente. El resultado de la suma se almacena en la variable “resultado\_suma”.

Las funciones y la modularidad son conceptos fundamentales en programación, ya que permiten escribir código más eficiente, fácil de entender y mantener. La práctica constante con estos conceptos te ayudará a mejorar tus habilidades de programación.

### 7.3. Módulos y su importación

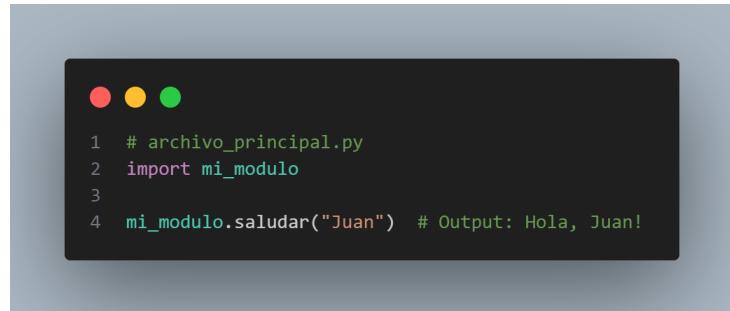
Los módulos son archivos que contienen definiciones y declaraciones de funciones, clases y variables en Python. Ayudan a organizar el código en archivos separados para hacerlo más legible y reutilizable. Para utilizar las funciones y variables definidas en un módulo en otro archivo, se necesita importar el módulo en el archivo donde se desea utilizar.

- Creación de un Módulo: Supongamos que tenemos un archivo llamado `mi_modulo.py` con la siguiente definición de función:



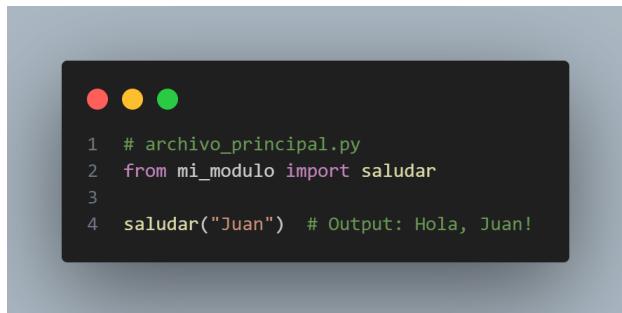
```
● ● ●
1 # mi_modulo.py
2
3 def saludar(nombre):
4     print("Hola, " + nombre + "!")
```

- Importación de un Módulo: En otro archivo Python, podemos importar y utilizar la función saludar del módulo mi\_modulo.py de la siguiente manera:



```
● ● ●
1 # archivo_principal.py
2 import mi_modulo
3
4 mi_modulo.saludar("Juan") # Output: Hola, Juan!
```

También se puede importar una función específica de un módulo para evitar usar el nombre del módulo cada vez que se llama a la función:



```
● ● ●
1 # archivo_principal.py
2 from mi_modulo import saludar
3
4 saludar("Juan") # Output: Hola, Juan!
```

La importación de módulos es esencial para organizar proyectos grandes y complejos en Python, ya que permite dividir el código en archivos manejables y fácilmente comprensibles. Además, facilita la reutilización del código, ya que las funciones y variables definidas en un módulo pueden ser utilizadas en varios archivos del proyecto.

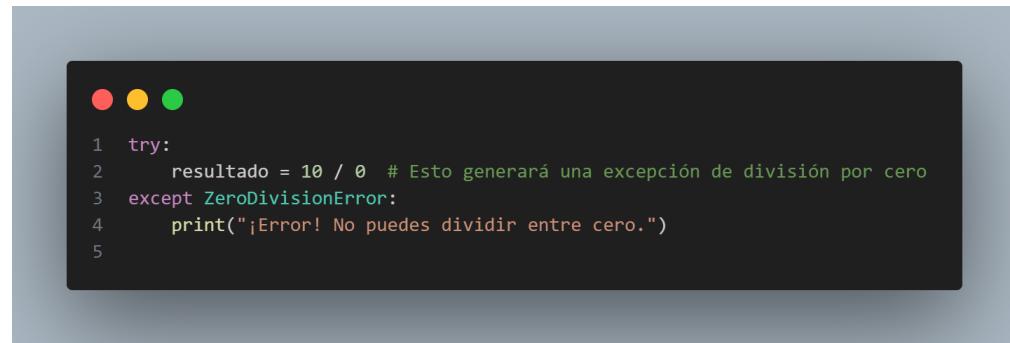
## 8. Manejo de excepciones y errores

### 8.1. Introducción a las excepciones

En Python, las excepciones son eventos inesperados o errores que ocurren durante la ejecución de un programa. Estos errores pueden ser causados por diversas situaciones, como divisiones por cero, acceso a archivos inexistentes o intentos de acceder a índices fuera de rango en listas. En resumen, Python nos proporciona una estructura para manejar estas excepciones y evitar que el programa se detenga abruptamente.

## 8.2. Uso de try, except, else

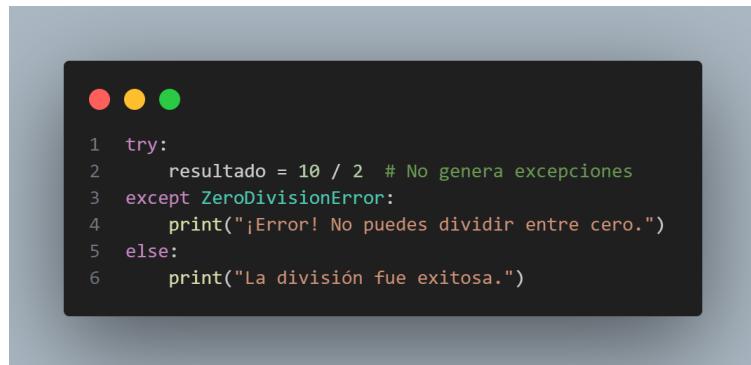
- Try: El bloque “try” se utiliza para envolver el código en el que esperas que ocurra una excepción. Dentro de este bloque, colocas el código que podría generar una excepción. Si una excepción se produce en el bloque “try”, el flujo de control se traslada al bloque “except”.
- Except: El bloque “except” se utiliza para manejar excepciones. Puedes especificar el tipo de excepción que deseas capturar después de la palabra clave “except”. Si ocurre una excepción del tipo especificado en el bloque try, se ejecutará el código en el bloque “except”. Puedes tener varios bloques “except” para manejar diferentes tipos de excepciones.



```
● ● ●

1 try:
2     resultado = 10 / 0 # Esto generará una excepción de división por cero
3 except ZeroDivisionError:
4     print("¡Error! No puedes dividir entre cero.")
5
```

- Else: El bloque “else” es opcional y se coloca después de todos los bloques “except”. Se ejecuta si no se ha producido ninguna excepción en el bloque “try”. Es útil para ejecutar código que debe ejecutarse solo si no se generan excepciones.

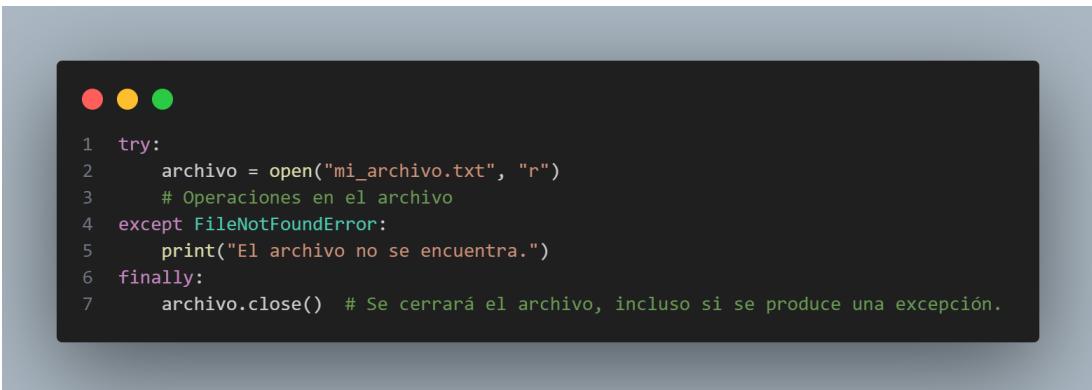


```
● ● ●

1 try:
2     resultado = 10 / 2 # No genera excepciones
3 except ZeroDivisionError:
4     print("¡Error! No puedes dividir entre cero.")
5 else:
6     print("La división fue exitosa.")
```

- Finally: El bloque “finally” es opcional y se coloca al final, después de todos los bloques “except” y, si se utiliza, después del bloque “else”. El código en este bloque se ejecuta sin importar si se produjo una excepción

o no. Se utiliza comúnmente para realizar la limpieza de recursos, como cerrar archivos o conexiones de bases de datos, garantizando que se realicen incluso en caso de excepciones.

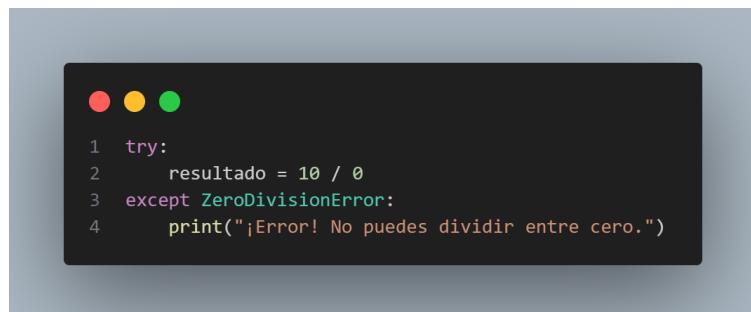


```
● ● ●

1 try:
2     archivo = open("mi_archivo.txt", "r")
3     # Operaciones en el archivo
4 except FileNotFoundError:
5     print("El archivo no se encuentra.")
6 finally:
7     archivo.close() # Se cerrará el archivo, incluso si se produce una excepción.
```

### 8.3. Tipos de excepciones y manejo específico

- ZeroDivisionError: Se genera cuando intentas dividir un número entre cero.



```
● ● ●

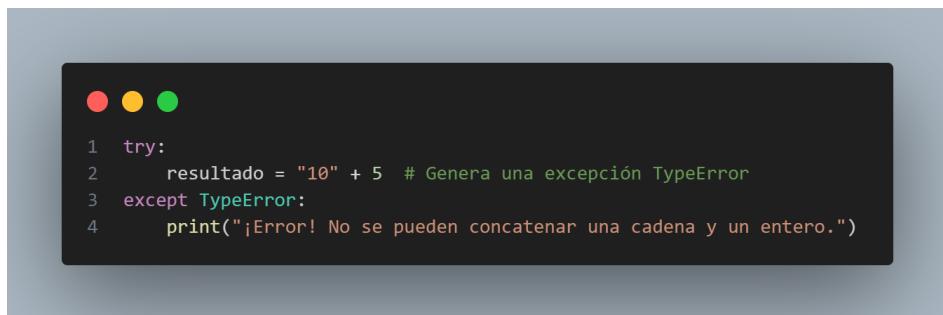
1 try:
2     resultado = 10 / 0
3 except ZeroDivisionError:
4     print("¡Error! No puedes dividir entre cero.")
```

- `FileNotFoundException`: Ocurre cuando intentas abrir o realizar operaciones en un archivo que no existe.



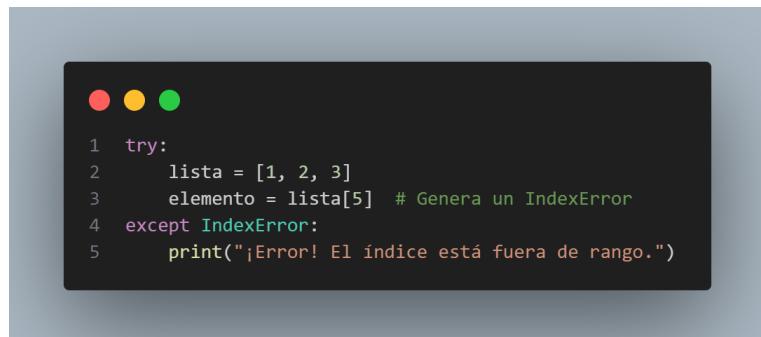
```
● ● ●  
1 try:  
2     archivo = open("mi_archivo.txt", "r")  
3     # Operaciones en el archivo  
4 except FileNotFoundError:  
5     print("El archivo no se encuentra.")
```

- `TypeError`: Esta excepción se produce cuando realizas una operación en un tipo de dato que no es compatible.



```
● ● ●  
1 try:  
2     resultado = "10" + 5  # Genera una excepción TypeError  
3 except TypeError:  
4     print("¡Error! No se pueden concatenar una cadena y un entero.")
```

- `IndexError`: Se genera al intentar acceder a un índice fuera de rango en una lista o secuencia.



```
● ● ●  
1 try:  
2     lista = [1, 2, 3]  
3     elemento = lista[5]  # Genera un IndexError  
4 except IndexError:  
5     print("¡Error! El índice está fuera de rango.")
```

- `ValueError`: Esta excepción ocurre cuando el tipo de dato es correcto, pero el valor no es válido para la operación.

```
● ● ●  
1 try:  
2     numero = int("abc") # Genera un ValueError  
3 except ValueError:  
4     print("¡Error! No se puede convertir 'abc' a un entero.")
```

- Custom Exceptions: Además de las excepciones incorporadas, puedes crear tus propias excepciones personalizadas extendiendo la clase Exception. Esto es útil cuando deseas manejar situaciones específicas en tu programa.

```
● ● ●  
1 class MiErrorPersonalizado(Exception):  
2     pass  
3  
4     try:  
5         raise MiErrorPersonalizado("Este es un error personalizado.")  
6     except MiErrorPersonalizado as error:  
7         print("Se ha producido un error personalizado:", error)
```

## 9. Entrada y salida de datos

### 9.1. Función print()

La función print() en Python es una herramienta fundamental para mostrar información en la consola o en la salida estándar del programa. Su propósito principal es facilitar a los programadores imprimir mensajes, variables, listas y otros datos que permiten observar el comportamiento del código.

- Salida de texto: print() Se utiliza para mostrar cadenas de textos en la consola. Esto es útil para comunicar mensajes, indicadores o información relevante durante la ejecución del programa.
- Imprimir variables: Imprimir variables es una práctica común en programación, ya que te permite observar el contenido de las variables en diferentes momentos de la ejecución de tu programa. Esto es esencial para el



```
● ● ●
1 print("holaaa")
2 >> holaa
```

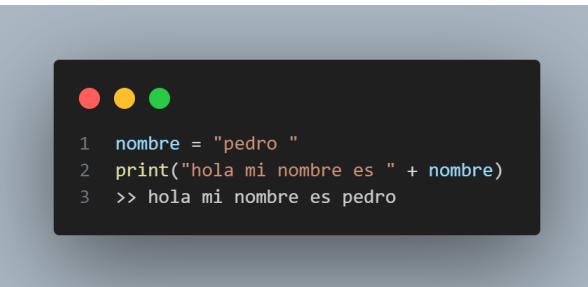
seguimiento del flujo de datos en tu código, a continuación se muestran algunos ejemplos.

- Ejemplo 1 : Para imprimir el valor de una variable, simplemente incluye a la variable como argumento de la función.



```
● ● ●
1 numero = 42
2 print(numero)
3 >> 42
```

- Ejemplo 2 : Concatenar variables de texto. Puedes combinar el valor de una variable con texto utilizando el operador de concatenación(+) . esto es útil para crear mensajes informativos



```
● ● ●
1 nombre = "pedro "
2 print("hola mi nombre es " + nombre)
3 >> hola mi nombre es pedro
```

## 9.2. Función input()

Ahora cómo enviar un dato mediante el teclado para que el programa lo tome en cuenta mediante el método input()

- Ejemplo 1



```
1 nombre = input("hola ¿cuál es tu nombre?----> ") #Benjamin
2 print("hola",nombre,"como estas ")
3 >> hola Benjamin, como estas
```

Esta función permite obtener el texto escrito por el usuario, al cual se le asignará un espacio a la memoria con la variable que el programador crea conveniente. Al llegar a la línea del comando, la consola esperara respuesta. Cuando el usuario escriba algo y presione la tecla Enter, el código seguirá ejecutándose.

Lo que estamos indicando al programa es que la variable «nombre» va a tomar el valor que el usuario ingrese cuando se le muestre el mensaje «Hola ¿Cuál es tu nombre?», para posteriormente, responder con otro mensaje y el valor que se ingresó. Debemos tener en cuenta que al usar `input()`, los datos ingresados siempre serán guardados como tipo string. Si necesitáramos ingresar números para utilizarlos en alguna operación matemática, debemos convertirlos a un tipo de dato adecuado (por ejemplo `int` o `float`, dependiendo si requerimos decimales). Podemos hacerlo de las siguientes maneras



```
1 edad = int(input("¿Que edad tienes?----> "))
2 print("Tu edad es",edad,"años ")
3 >> Tu edad es 17 años
```

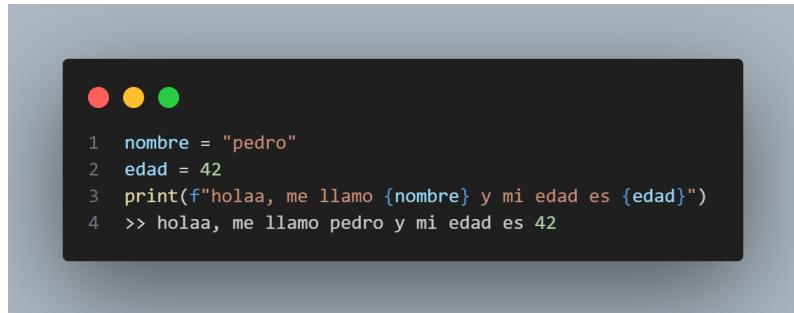
### 9.3. Formateo de cadenas

Python ofrece varias formas de formatear cadenas, y dos de los más comunes son mediante f-strings (cadenas formateadas) y métodos de formatos.

- Utilizan la sintaxis de cadenas con una “f” o “F”
- Permiten incrustar expresiones o variables mediante llaves “{”
- Pueden realizar cambios y formateos dentro de la llave.
- Son una forma más legibles y eficiente de formatear cadenas en python.

En python, una cadena de texto normal se escribe entre comillas (""), para crear f-strings solo tienes que agregar la letra f o F mayúscula antes de la comilla.

¿Cómo imprimir variables usando f-strings?

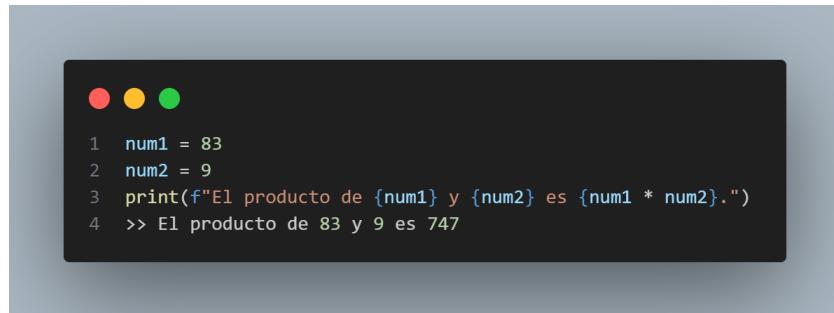


```
● ● ●
1 nombre = "pedro"
2 edad = 42
3 print(f"holaa, me llamo {nombre} y mi edad es {edad}")
4 >> holaa, me llamo pedro y mi edad es 42
```

¿Cómo evaluar expresiones en python con f-string?

Como las f-Strings son evaluadas al momento de ejecución (cuando se está ejecutando el código Python), bien podrías usar una f-string para evaluar expresiones válidas.

En el siguiente ejemplo. num1 y num2 son nuestras variables, para obtener el producto de estas variables únicamente basta con escribir la siguiente expresión num1 \* num2 dentro de llaves



```
● ● ●
1 num1 = 83
2 num2 = 9
3 print(f"El producto de {num1} y {num2} es {num1 * num2}.")
4 >> El producto de 83 y 9 es 747
```

En cualquier f-string, variable nombre y expresión se sustituirán por los valores que representan o por el valor resultante de una operación en el momento de ejecución (cuando se está ejecutando el código Python)

## 10. Trabajo con archivos

### 10.1. Apertura, lectura y escritura de archivos

Para trabajar con archivos en Python, se utiliza la función open(). Esta función toma dos argumentos: el nombre del archivo y el modo de apertura.

El modo de apertura determina cómo se puede acceder al archivo. Los modos de apertura más comunes son:

- r: Lectura. El archivo se abre en modo de lectura.
- w: Escritura. El archivo se abre en modo de escritura. Si el archivo no existe, se crea uno nuevo. Si el archivo existe, se sobrescribe su contenido.
- a: Adición. El archivo se abre en modo de adición. Los datos se escriben al final del archivo.

Por ejemplo, para abrir un archivo de texto en modo de lectura, se puede usar el siguiente código:

Una vez que un archivo está abierto, se puede leer su contenido usando la función `read()`. Esta función devuelve una cadena con todo el contenido del archivo.

Por ejemplo, para leer el contenido de un archivo de texto, se puede usar el siguiente código:

También se puede leer el contenido de un archivo de texto línea por línea usando un bucle `for`:

Para escribir datos en un archivo, se usa la función `write()`. Esta función toma una cadena como argumento y la escribe al final del archivo.

Por ejemplo, para escribir el siguiente texto en un archivo de texto:  
Se puede usar el siguiente código:

## 10.2. Operaciones comunes de archivos

Además de la apertura, lectura y escritura, Python ofrece otras operaciones comunes de archivos.

- `seek()`: Permite mover el puntero de lectura o escritura a una posición determinada en el archivo.
- `tell()`: Devuelve la posición actual del puntero de lectura o escritura en el archivo.
- `flush()`: Escribe cualquier dato que aún no se haya escrito en el archivo al disco.
- `close()`: Cierra el archivo y libera los recursos asociados con él.

## 10.3. Manejo de archivos con el bloque “with”

El bloque `with` es una forma segura y conveniente de abrir y cerrar archivos. Cuando se usa el bloque `with`, el archivo se abre automáticamente al principio del bloque y se cierra automáticamente al final del bloque.

Por ejemplo, el siguiente código abre un archivo de texto en modo de lectura y luego imprime su contenido:

Por ejemplo, el siguiente código abre un archivo de texto en modo de lectura y luego imprime su contenido:

El uso del bloque `with` evita errores comunes, como olvidar cerrar un archivo abierto.



- 11. Introducción a la programación orientada a objetos**
  - 11.1. Definición de clases y objetos**
  - 11.2. Métodos y atributos de clase**
- 12. Clases y objetos**
  - 12.1. Encapsulación y visibilidad**
  - 12.2. Métodos especiales**
  - 12.3. Herencia y composición**
- 13. Herencia y polimorfismo**
  - 13.1. Herencia simple y múltiple**
  - 13.2. Polimorfismo y sobrecarga de métodos**
  - 13.3. Clases abstractas e interfaces**
- 14. Módulos y paquetes**
  - 14.1. ¿Qué son los Módulos en Python?**
  - 14.2. Creación y Uso de Módulos**
  - 14.3. Importación de Módulos**
  - 14.4. Módulos Estándar de Python**
  - 14.5. ¿Qué son los Paquetes en Python?**
  - 14.6. Creación y Organización de Paquetes**
  - 14.7. Importación de Paquetes**
- 15. Manipulación de strings y expresiones regulares**
  - 15.1. Operaciones comunes con cadenas de texto**
  - 15.2. Operaciones comunes con cadenas de texto**
- 16. Trabajo con fechas y tiempos**
  - 16.1. Módulos datetime y time**
  - 16.2. Operaciones y formateo de fechas y tiempos**
- 17. Introducción a Git**
  - 17.1. ¿Qué es Git y por qué es importante?**
  - 17.2. Configuración inicial de Git**
  - 17.3. Comandos básicos de Git**