# Sentencias básicas del lenguaje Python

---

**Módulo:** Fundamentos de programación Python para el análisis de datos.

**AE2**: Aplicar el concepto de variable, tipos de dato fundamentales y expresiones aritméticas utilizando el lenguaje python para la creación de una rutina de baja complejidad.

alkemy

# Introducción 🧑‍💻

Python es un lenguaje de programación conocido por su simplicidad y claridad, lo que lo hace ideal tanto para principiantes como para programadores experimentados. Al aprender las sentencias básicas de Python, los nuevos desarrolladores adquieren las habilidades fundamentales para manipular datos, realizar cálculos, manejar el flujo de datos en un programa y obtener interacción con el usuario. Estos fundamentos son la base sobre la cual se construyen aplicaciones más complejas y avanzadas.

Este manual cubre los conceptos esenciales de Python, incluyendo el uso de variables, tipos de datos, expresiones aritméticas, conversiones de tipo y las interacciones con el usuario a través de la consola. Además, exploraremos cómo crear y ejecutar scripts en Python, lo que permite desarrollar programas que ejecutan múltiples operaciones y gestionan datos de forma eficiente. Con una comprensión sólida de estos conceptos, los desarrolladores pueden avanzar hacia proyectos más complejos en Python y aprovechar la versatilidad del lenguaje en una amplia variedad de aplicaciones.

## Aprendizaje esperado

Cuando finalices la lección serás capaz de:

- Definir y utilizar variables en Python para almacenar y manipular datos.
- Identificar y trabajar con los tipos de datos fundamentales en Python, como enteros, decimales, cadenas de caracteres y booleanos.
- Crear expresiones aritméticas para realizar cálculos y operar con datos numéricos.
- Realizar conversiones de tipo para adaptar datos a diferentes formatos según las necesidades del programa.
- Utilizar las sentencias de impresión y entrada de datos en consola para interactuar con el usuario.
- Crear y ejecutar scripts en Python, consolidando las bases para desarrollar programas funcionales.

# Desarrollo

## Sección 1: Variables

### Definición y Uso de Variables

En Python, una variable es un contenedor para almacenar datos que pueden utilizarse y manipularse en un programa. Las variables permiten asignar valores a un nombre específico, lo que facilita su referencia y manipulación en operaciones posteriores. A diferencia de otros lenguajes, Python no requiere declarar el tipo de una variable antes de usarla, ya que es un lenguaje de tipado dinámico. Esto significa que el tipo de datos de una variable se determina automáticamente cuando se le asigna un valor.

Para asignar un valor a una variable, se utiliza el operador de asignación `=`, de modo que el nombre de la variable se coloca a la izquierda del signo igual y el valor asignado a la derecha. Por ejemplo, `edad = 25` asigna el valor 25 a la variable `edad`. Las variables pueden almacenar diferentes tipos de datos, como números, cadenas de caracteres y valores booleanos, y se pueden cambiar o reasignar en cualquier momento.

Ejemplo de asignación de variables:

```
edad = 25  # Almacena el valor 25 en la variable 'edad'
nombre = "Juan"  # Almacena la cadena "Juan" en la variable 'nombre'
pi = 3.1416  # Almacena el valor decimal en 'pi'
```

Estas variables pueden usarse posteriormente en cualquier operación o impresión, como en:

```
print("La edad de", nombre, "es", edad)
# Salida: La edad de Juan es 25
```

### Buenas Prácticas al Nombrar Variables

Elegir nombres descriptivos y coherentes para las variables es una buena práctica que facilita la lectura y el mantenimiento del código. En Python, es común utilizar nombres de variables en minúsculas y separados por guiones bajos (_) cuando el nombre contiene más de una palabra, como en `edad_usuario`. Evitar el uso de nombres de variables muy cortos o genéricos

ayuda a evitar confusiones y errores en el código, especialmente en proyectos grandes o colaborativos.

Ejemplo de buenos nombres de variables:

```
altura_persona = 1.75
salario_anual = 50000
nombre_empresa = "TechCorp"
```

**Variables Dinámicas en Python**

Como Python es un lenguaje de tipado dinámico, una variable puede cambiar de tipo durante la ejecución del programa. Por ejemplo, una variable que inicialmente contiene un número puede cambiarse para almacenar una cadena de texto más adelante. Aunque esta flexibilidad facilita el desarrollo rápido, se debe tener cuidado al cambiar el tipo de una variable para evitar errores en el programa.

Ejemplo de cambio de tipo de una variable:

```
dato = 10  # Inicialmente, 'dato' es un entero
print(dato)  # Salida: 10

dato = "Diez"  # Ahora, 'dato' es una cadena
print(dato)  # Salida: Diez
```

**Scope o Alcance de las Variables**

El alcance de una variable se refiere a la parte del programa donde la variable es accesible. En Python, una variable definida dentro de una función es local y solo puede utilizarse dentro de esa función. Por otro lado, una variable definida fuera de una función tiene un alcance global y puede accederse desde cualquier parte del programa. Es importante gestionar el alcance de las variables para evitar conflictos y asegurar la coherencia de los datos.

**Declaración Múltiple de Variables**

Python permite asignar valores a varias variables en una sola línea, lo que puede ser útil en situaciones donde se deben definir múltiples variables al mismo tiempo. Por ejemplo, `x, y, z = 1, 2, 3` asigna los valores 1, 2 y 3 a `x`, `y` y `z`

respectivamente. Esta sintaxis hace que el código sea más compacto y legible cuando se trabaja con múltiples variables de manera simultánea.

Ejemplo de asignación múltiple:

```
x, y, z = 1, 2, 3
print(x, y, z)  # Salida: 1 2 3

# También se pueden asignar el mismo valor a varias variables
a = b = c = 0
print(a, b, c)  # Salida: 0 0 0
```

## Sección 2: Tipos de Dato Fundamentales

### Enteros

Los números enteros son uno de los tipos de datos más básicos en Python, representados por el tipo `int`. Los enteros permiten almacenar valores numéricos sin decimales, como `-5`, `0`, y `100`. Los enteros en Python tienen un rango amplio y no están limitados en tamaño, lo que permite trabajar con números extremadamente grandes sin necesidad de tipos adicionales. Este tipo de dato es útil en cálculos matemáticos básicos y en situaciones donde se necesita contar o iterar en bucles.

Ejemplo de operaciones con enteros:

```
x = 10
y = -5
suma = x + y
print("Suma:", suma)  # Salida: 5
```

### Decimales

Los números decimales, o de punto flotante, están representados por el tipo `float` en Python y permiten almacenar números con decimales, como `3.14`, `-2.5`, y `0.001`. Los `float` son útiles cuando se requiere precisión en cálculos que involucran fracciones o porcentajes. Python también ofrece el módulo `decimal` para manejar decimales con mayor precisión, lo cual es útil en aplicaciones financieras o científicas donde la precisión es crucial.

Ejemplo de uso de decimales:

```python
precio = 19.99
descuento = 0.15
precio_final = precio * (1 - descuento)
print("Precio final:", precio_final)  # Salida: 16.9915
```

**Cadena de Caracteres**

Las cadenas de caracteres, o `str`, representan secuencias de texto y son ampliamente utilizadas para almacenar y manipular datos textuales. En Python, las cadenas pueden definirse entre comillas simples (`'texto'`) o dobles (`"texto"`), y se pueden concatenar, dividir y manipular con facilidad. Python ofrece una serie de métodos para trabajar con cadenas, como `upper()`, `lower()`, y `replace()`, que permiten transformar y modificar el texto de forma eficiente.

Ejemplo de uso de cadenas:

```python
nombre = "juan perez"
nombre_mayusculas = nombre.upper()
print(nombre_mayusculas)  # Salida: JUAN PEREZ
```

**Booleanos**

El tipo de dato booleano (`bool`) representa valores de verdad y puede ser `True` o `False`. Los booleanos son fundamentales en Python, especialmente en estructuras de control como las sentencias `if` y `while`, ya que permiten tomar decisiones basadas en condiciones. El valor booleano es el resultado de expresiones lógicas, como comparaciones entre números o cadenas de caracteres.

Ejemplo de comparación y valor booleano:

```python
edad = 18
es_mayor = edad >= 18
print("¿Es mayor de edad?", es_mayor)  # Salida: ¿Es mayor de edad? True
```

**Conversión entre Tipos de Datos**

Python permite convertir entre tipos de datos utilizando funciones de conversión como `int()`, `float()`, `str()`, y `bool()`. La conversión es útil cuando se necesita adaptar un tipo de datos específico para una operación o para mejorar

la coherencia en el programa. Por ejemplo, si se necesita concatenar un número con una cadena, se puede convertir el número a cadena utilizando `str(numero)`. Esta flexibilidad en la conversión facilita la manipulación de datos en Python.

Ejemplo de conversión de tipos:

```
edad_texto = "25"
edad_numero = int(edad_texto)
print("Edad como número:", edad_numero)  # Salida: Edad como número: 25
```

## Sección 3: Expresiones Aritméticas

### Operadores Aritméticos Básicos

Python permite realizar operaciones matemáticas básicas mediante operadores aritméticos como `+` para la suma, `-` para la resta, `*` para la multiplicación, y `/` para la división. Estos operadores se pueden usar con variables y valores constantes para crear expresiones aritméticas que facilitan la manipulación de datos numéricos. Las operaciones se ejecutan siguiendo el orden de operaciones, lo que asegura la coherencia de los cálculos.

Ejemplo de operaciones aritméticas:

```
a = 10
b = 3
suma = a + b
resta = a - b
multiplicacion = a * b
division = a / b
print("Suma:", suma, "Resta:", resta, "Multiplicación:", multiplicacion
```

### Operador de Módulo y División Entera

Además de los operadores básicos, Python incluye el operador de módulo (`%`), que devuelve el residuo de una división. Esto es útil en casos donde se necesita determinar si un número es divisible por otro, como en la verificación de números pares o impares. Python también permite realizar divisiones enteras utilizando `//`, que devuelve solo la parte entera del resultado, sin el decimal. Estas operaciones son útiles en situaciones que requieren precisión en el manejo de divisores y múltiplos.

Ejemplo de uso de módulo y división entera:

```
dividendo = 10
divisor = 3
residuo = dividendo % divisor
division_entera = dividendo // divisor
print("Residuo:", residuo)  # Salida: Residuo: 1
print("División entera:", division_entera)  # Salida: División entera: 3
```

**Operador de Exponenciación**

El operador `**` en Python permite calcular potencias, como `3**2` para elevar 3 al cuadrado. La exponenciación es común en problemas matemáticos y científicos, y Python permite realizar esta operación con alta precisión. Además, la biblioteca matemática (`math`) de Python ofrece funciones avanzadas, como el cálculo de raíces y logaritmos, que permiten realizar operaciones matemáticas complejas de manera eficiente.

Ejemplo de exponenciación:

```
base = 2
exponente = 3
potencia = base ** exponente
print("Potencia:", potencia)  # Salida: Potencia: 8
```

**Operaciones Compuestas**

Python permite combinar varias operaciones en una misma expresión aritmética, respetando el orden de operaciones (PEMDAS: paréntesis, exponentes, multiplicación y división, y finalmente suma y resta). Esto facilita la creación de expresiones complejas en una sola línea de código, lo que puede ser útil en aplicaciones que requieren cálculos matemáticos elaborados.

Ejemplo de operaciones compuestas:

```
resultado = (2 + 3) * 4 - 5 / 2
print("Resultado:", resultado)
```

**Uso de Paréntesis para Controlar el Orden de Operaciones**

El uso de paréntesis en Python permite definir el orden de las operaciones en una expresión aritmética. Esto es particularmente útil para evitar errores y asegurar que los cálculos se realicen de la manera deseada. Por ejemplo, en la expresión `(2 + 3) * 4`, los paréntesis aseguran que la suma se realice antes de la

multiplicación. Controlar el orden de operaciones es crucial para evitar resultados inesperados y garantizar la precisión de los cálculos.

Ejemplo de uso de paréntesis:

```
resultado = 2 + 3 * 4  # Sin paréntesis, multiplica primero
print("Resultado sin paréntesis:", resultado)  # Salida: 14

resultado = (2 + 3) * 4  # Con paréntesis, suma primero
print("Resultado con paréntesis:", resultado)  # Salida: 20
```

## Sección 4: Conversiones de Tipo

### Conversión de Enteros a Flotantes

Python permite convertir enteros a números de punto flotante utilizando la función `float()`. Esta conversión es útil en casos donde se necesita precisión decimal, como en cálculos financieros o científicos. Convertir un entero a flotante también evita posibles errores en operaciones que requieren decimales, como la división, asegurando que el resultado se ajuste al contexto.

Ejemplo de conversión de entero a flotante:

```
entero = 10
decimal = float(entero)
print("Convertido a decimal:", decimal)  # Salida: Convertido a decimal: 10.0
```

### Conversión de Flotantes a Enteros

Para convertir un número decimal a entero, se utiliza la función `int()`. Esta conversión trunca los decimales, lo que puede ser útil en situaciones donde se necesita redondear hacia abajo un valor. Sin embargo, es importante recordar que la conversión de `float` a `int` implica una pérdida de precisión, ya que el valor decimal se elimina. En aplicaciones que requieren redondeo, Python ofrece funciones como `round()` para redondear al entero más cercano.

Ejemplo de conversión de decimal a entero:

```python
numero = 15.7
entero = int(numero)
print("Convertido a entero:", entero)  # Salida: Convertido a entero: 15
```

**Conversión de Números a Cadenas y Viceversa**

La conversión entre números y cadenas es común en Python, especialmente cuando se necesita manipular datos textuales que contienen valores numéricos. La función `str()` convierte un número en una cadena de texto, permitiendo su concatenación con otras cadenas. Para convertir una cadena a un número, se utilizan las funciones `int()` o `float()`, dependiendo de si se necesita un número entero o decimal.

Ejemplo de conversión de número a cadena:

```python
numero = 123
texto = str(numero)
print("Número como texto:", texto)  # Salida: Número como texto: 123
```

**Conversión de Cadenas a Booleanos**

La función `bool()` convierte una cadena de caracteres en un valor booleano. En Python, las cadenas vacías (`""`) se consideran `False`, mientras que cualquier otra cadena se considera `True`. Esta conversión es útil en estructuras de control, donde se necesita verificar la existencia de un valor en una cadena o en operaciones que involucren valores condicionales.

Ejemplo de conversión de cadena a booleano:

```python
cadena = "Python"
es_verdadero = bool(cadena)
print("¿Es verdadero?", es_verdadero)  # Salida: ¿Es verdadero? True
```

**Conversión Implícita en Operaciones**

Python realiza conversiones de tipo implícitas en algunas operaciones, como al sumar un entero y un flotante, donde el entero se convierte automáticamente en flotante para evitar errores. Esta característica hace que el código sea más flexible, aunque es importante entender cómo Python realiza estas conversiones para evitar resultados inesperados. Conocer y aprovechar las conversiones

implícitas permite escribir código que se adapta automáticamente al tipo de datos involucrado.

Ejemplo de conversión implícita:

```
entero = 5
decimal = 2.5
resultado = entero + decimal
print("Resultado:", resultado)  # Salida: Resultado: 7.5
```

## Sección 5: Impresión en Consola

### Función `print()` para Mostrar Información

La función `print()` en Python permite mostrar información en la consola, lo que es útil para presentar resultados, mensajes al usuario y el seguimiento de la ejecución del programa. Esta función es fundamental para depurar el código y comunicar información, y es ampliamente utilizada en la fase de desarrollo para verificar el comportamiento del programa.

Ejemplo de `print()` básico:

```
print("Hola, Mundo!")
```

### Impresión de Variables y Textos Combinados

Python permite combinar variables y textos en una sola sentencia `print()` utilizando la concatenación de cadenas o las técnicas de formato. Con la concatenación, se puede unir texto y variables utilizando el operador `+`. Sin embargo, en Python es más común utilizar `f-strings` o el método `format()` para combinar texto y variables de manera elegante y legible, como en `print(f"El resultado es {resultado}")`.

Ejemplo de `f-string` para impresión combinada:

```
nombre = "Ana"
edad = 30
print(f"{nombre} tiene {edad} años.")  # Salida: Ana tiene 30 años.
```

### Control de la Separación y el Final de la Impresión

La función `print()` permite controlar el separador entre elementos y el carácter final de la impresión. Por ejemplo, `print("Hola", "Mundo", sep="-")` imprime "Hola-Mundo" en lugar de agregar un espacio entre las palabras. También se puede utilizar el parámetro `end` para definir el carácter final, como en `print("Hola", end="!")`, lo que evita que se agregue una nueva línea al final de la impresión.

Ejemplo:

```
print("Python", "es", "genial", sep="-", end="!")  # Salida: Python-es-genial!
```

**Impresión de Formato Avanzado**

Python permite imprimir en formatos avanzados utilizando la interpolación de cadenas, `f-strings` y el método `format()`. Esto es útil para controlar la cantidad de decimales en números flotantes, alinear texto y presentar datos de manera organizada. La impresión de formato avanzado es especialmente útil en aplicaciones que presentan datos al usuario, ya que facilita la legibilidad y organización de la información.

Ejemplo de impresión avanzada:

```
precio = 49.99
print(f"El precio es: ${precio:.2f}")  # Salida: El precio es: $49.99
```

**Usos Prácticos de `print()` en Depuración**

La función `print()` es una herramienta valiosa para la depuración, ya que permite verificar el valor de variables en diferentes partes del programa y rastrear el flujo de ejecución. Al imprimir el estado de las variables en momentos clave, los programadores pueden identificar problemas y ajustar el código. Esto es especialmente útil en etapas iniciales del desarrollo, donde el seguimiento y la retroalimentación inmediata son esenciales.

Ejemplo de depuración con `print()`:

```
contador = 0
for i in range(5):
    contador += i
    print("Contador:", contador)  # Salida en cada iteración
```

## Sección 6: Entrada de Datos en Consola

### Función `input()` para Capturar Datos del Usuario

La función `input()` permite capturar datos ingresados por el usuario desde la consola, convirtiéndolos automáticamente en una cadena de texto. Esta funcionalidad es esencial en aplicaciones interactivas, ya que permite personalizar la ejecución del programa según las respuestas del usuario. Por ejemplo, `nombre = input("¿Cuál es tu nombre? ")` solicita el nombre y lo almacena en la variable `nombre`.

Ejemplo de `input()`:

```
nombre = input("¿Cuál es tu nombre? ")
print("Hola,", nombre)
```

### Conversión de Entradas de Texto a Otros Tipos de Datos

Como `input()` siempre devuelve una cadena, es común convertir los datos ingresados a otros tipos, como enteros o decimales, para realizar cálculos o comparaciones. Por ejemplo, se puede utilizar `int(input("Ingresa tu edad: "))` para capturar la edad como un número entero, permitiendo realizar operaciones aritméticas con el valor ingresado.

Ejemplo de conversión de `input()` a número:

```
edad = int(input("¿Cuántos años tienes? "))
print("Tendrás", edad + 1, "años el próximo año.")
```

### Manejo de Errores en Entradas de Datos

Al capturar entradas del usuario, es importante anticipar posibles errores, como el ingreso de texto cuando se espera un número. Python permite manejar estos errores mediante sentencias `try` y `except`, lo que garantiza que el programa no se detenga abruptamente y que el usuario reciba mensajes de corrección. Esta validación de entradas es crucial en aplicaciones que requieren datos específicos.

Ejemplo de manejo de errores:

```python
try:
    numero = int(input("Ingresa un número: "))
    print("El doble es:", numero * 2)
except ValueError:
    print("Por favor, ingresa un número válido.")
```

#### Uso de Prompts y Mensajes Claros

El mensaje de `input()` (prompt) debe ser claro para que el usuario entienda qué se espera de él. Al proporcionar indicaciones claras y precisas, se reduce la probabilidad de errores en la entrada de datos. Por ejemplo, "Ingrese su edad en años:" es más claro que solo "Edad:", y ayuda al usuario a proporcionar los datos correctos desde el principio.

Ejemplo de prompt claro:

```python
edad = input("Ingresa tu edad en años: ")
```

#### Interactividad en Programas Simples

La función `input()` permite crear programas interactivos básicos que se adaptan a las respuestas del usuario. Esto es útil para desarrollar encuestas, juegos, calculadoras y otras aplicaciones que dependen de la participación del usuario. La interactividad es fundamental en muchas aplicaciones modernas, y `input()` permite implementar esta característica de forma simple en Python.

Ejemplo:

```python
nombre = input("¿Cuál es tu nombre? ")
print(f"¡Bienvenido, {nombre}!")
```

### Sección 7: Creación y Ejecución de un Script Python

#### Estructura de un Script en Python

Un script en Python es un archivo que contiene código Python, generalmente con la extensión `.py`, que se puede ejecutar directamente en el intérprete. Los scripts permiten agrupar múltiples instrucciones en un solo archivo, lo que facilita la ejecución y reutilización del código. La estructura de un script puede incluir variables, funciones y sentencias de control que interactúan para realizar tareas específicas.

Ejemplo de script básico:

```
# archivo: mi_script.py
nombre = input("¿Cuál es tu nombre? ")
print("Hola,", nombre)
```

**Ejecutar un Script desde la Consola**

Para ejecutar un script en Python, basta con abrir la terminal y escribir `python nombre_del_script.py`, donde `nombre_del_script.py` es el archivo que contiene el código. Esto permite ejecutar programas complejos y automáticos sin necesidad de ejecutar cada línea manualmente. Ejecutar scripts desde la consola es especialmente útil en aplicaciones automatizadas y procesos que requieren múltiples pasos consecutivos.

**Ejecutar un Script desde la Consola**

Para ejecutar, abre la terminal y escribe:

```
python mi_script.py
```

**Beneficios de Usar Scripts para Automatización**

Los scripts de Python permiten automatizar tareas repetitivas, como el procesamiento de datos, el manejo de archivos o la ejecución de cálculos periódicos. Esto hace que los scripts sean una herramienta valiosa en áreas como la administración de sistemas, la ciencia de datos y la automatización de procesos en general. La posibilidad de programar y ejecutar scripts de forma automática también facilita la integración de Python en entornos de trabajo más complejos.

Ejemplo de script para sumar números:

```
# archivo: sumar.py
a = int(input("Primer número: "))
b = int(input("Segundo número: "))
print("La suma es:", a + b)
```

**Incorporación de Parámetros y Argumentos en un Script**

Python permite pasar parámetros a los scripts mediante argumentos de línea de comandos, utilizando la biblioteca `argparse`. Esto permite personalizar la

ejecución del script sin modificar el código. Por ejemplo, `python script.py --nombre Juan` permite ejecutar el script con el argumento `nombre` igual a "Juan", lo que hace que el script sea flexible y adaptable a diferentes escenarios.

Ejemplo:

```python
# archivo: saludo.py
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--nombre", required=True)
args = parser.parse_args()
print(f"¡Hola, {args.nombre}!")
```

# Cierre ✍️

Este manual ha introducido los conceptos fundamentales de Python, incluyendo la definición y uso de variables, los tipos de datos básicos, las expresiones aritméticas y la interacción con el usuario a través de la consola. Estos conocimientos son la base sobre la cual se construyen programas más complejos y ofrecen una comprensión clara de cómo se estructura y manipula la información en Python. La posibilidad de crear y ejecutar scripts en Python también permite que los desarrolladores automaticen tareas y desarrollen aplicaciones que interactúan de manera efectiva con el usuario.

Con una base sólida en estas sentencias básicas, los estudiantes pueden avanzar hacia conceptos más complejos, como estructuras de control, funciones y manejo de errores, consolidando así su habilidad para desarrollar soluciones en Python. La práctica y la aplicación de estos conocimientos son esenciales para el aprendizaje continuo y para dominar el lenguaje Python en proyectos de mayor escala.

# Referencias📚

Python Software Foundation. (s.f.). *Python 3 documentation*. https://docs.python.org/3/

Real Python. (s.f.). *Learn Python programming*. https://realpython.com/

W3Schools. (s.f.). *Python tutorial*. https://www.w3schools.com/python/

# ¡Muchas gracias!

Nos vemos en la próxima lección