1.Introducción a la Programación

1.1. ¿Qué es la programación?

La **programación** es la disciplina encargada de diseñar, escribir y mantener el código fuente que controla el comportamiento de las computadoras. Es un componente esencial de la informática, y permite crear software que resuelve problemas, automatiza tareas y optimiza procesos.

A través de la **programación**, se transforman ideas en soluciones computacionales, empleando lenguajes específicos que permiten a los programadores comunicarse con el hardware. Esta ciencia combina lógica, matemáticas, y creatividad, permitiendo desarrollar desde aplicaciones sencillas como calculadoras hasta sistemas complejos como los que operan autos autónomos o gestionan redes sociales.

La **programación** también es el motor detrás del desarrollo tecnológico moderno, permitiendo la innovación en áreas como la inteligencia artificial, el análisis de grandes volúmenes de datos (big data), y el internet de las cosas (IoT). Aprender a programar no solo otorga la capacidad de crear software, sino que también fomenta habilidades de pensamiento crítico, resolución de problemas, y abstracción, todas ellas esenciales en un mundo cada vez más digital.

A su vez es el proceso de crear instrucciones que le indican a una computadora cómo realizar tareas específicas. Estas instrucciones se escriben en lenguajes de programación, que actúan como puentes entre el pensamiento humano y el funcionamiento de las máquinas. Python es un lenguaje de programación de alto nivel, conocido por su sintaxis clara y legible, lo que lo convierte en una excelente opción para principiantes.

1.2. Programas y algoritmos

Para entender la relación entre ambos primero se necesitaría saber, de manera breve, que es cada uno.

Un **programa** es un conjunto de instrucciones que una computadora sigue para realizar tareas específicas. Estas instrucciones son escritas en un lenguaje de programación, como Python, y traducidas a un formato que la computadora puede entender y ejecutar.

Cada programa tiene un propósito, ya sea resolver un problema, automatizar procesos o simplemente proporcionar una interfaz para interactuar con el hardware. En esencia, un programa es la implementación concreta de un algoritmo, utilizando los recursos del sistema como la memoria, el procesador y los dispositivos de entrada/salida para realizar las operaciones indicadas.

Un **algoritmo** es una secuencia finita y ordenada de pasos lógicos que resuelven un problema o realizan una tarea. Cada paso en un algoritmo describe una acción que debe ejecutarse en un orden específico para alcanzar el resultado deseado. Los algoritmos no están ligados a ningún lenguaje de programación; en su forma más básica, son descripciones abstractas de soluciones que luego se pueden implementar en cualquier lenguaje.

Ejemplo de un Algoritmo (Instrucciones Cotidianas)

Imaginemos que queremos hacer un sándwich. El algoritmo para esta tarea sería algo así:

1. Tomar dos rebanadas de pan.
2. Colocar una rebanada de jamón sobre una de las rebanadas de pan.
3. Poner una rebanada de queso encima del jamón.
4. Añadir condimentos al gusto (mostaza, mayonesa, etc.).
5. Colocar la otra rebanada de pan encima del queso.
6. Cortar el sándwich por la mitad (opcional).
7. Servir en un plato.

Este conjunto de pasos es un algoritmo porque, siguiendo cada instrucción en el orden indicado, siempre se logra el objetivo final: hacer un sándwich. De manera similar, en programación, un algoritmo define los pasos necesarios para resolver un problema, como ordenar una lista de números o buscar información en una base de datos.

Relación entre Programa y Algoritmo

El algoritmo es el corazón de cualquier programa. Primero, se define el algoritmo que resuelve el problema de manera eficiente y luego, se traduce ese algoritmo en un programa utilizando un lenguaje de programación como Python. En otras palabras, un programa es la "traducción" de un algoritmo en una forma que la computadora puede procesar.

1.3. Conceptos fundamentales: entrada, proceso y salida

Todo programa informático se basa en tres componentes esenciales:

- **Entrada**: Datos o información que el programa recibe para procesar.
- **Proceso**: Operaciones que el programa realiza sobre los datos de entrada.
- **Salida**: Resultados generados después del procesamiento.

Por ejemplo, en un programa que calcula la suma de dos números:

- **Entrada**: Los dos números proporcionados por el usuario.
- **Proceso**: La suma de esos dos números.
- **Salida**: El resultado de la suma.

1.4. Lenguajes de programación

Los lenguajes de programación son herramientas que permiten a los desarrolladores comunicarse con las computadoras. Se dividen en varias categorías:

1.4.1. Lenguaje máquina y ensamblador

- **Lenguaje máquina**: Es el lenguaje más básico, compuesto por códigos binarios (0s y 1s) que la computadora entiende directamente. Es difícil de leer y escribir para los humanos.

- **Lenguaje ensamblador**: Es una representación más legible del lenguaje máquina, utilizando abreviaturas y símbolos. Aunque es más comprensible que el lenguaje máquina, sigue siendo complejo y específico para cada tipo de procesador.

1.4.2. Lenguajes de alto nivel

Son lenguajes más cercanos al lenguaje humano, diseñados para ser fáciles de leer y escribir. Permiten a los programadores escribir instrucciones sin preocuparse por los detalles del hardware subyacente. Python es un ejemplo destacado de un lenguaje de alto nivel.

1.4.3. Lenguajes compilados vs. interpretados

- **Lenguajes compilados**: El código fuente se traduce completamente a lenguaje máquina antes de ejecutarse, mediante un programa llamado compilador. Ejemplos incluyen C y C++.
- **Lenguajes interpretados**: El código fuente se traduce y ejecuta línea por línea en tiempo real, utilizando un intérprete. Python es un lenguaje interpretado, lo que facilita la prueba y depuración del código.

1.4.4. Proceso de compilación

El proceso de compilación es el conjunto de pasos que transforman el código fuente en un programa ejecutable. Este proceso implica varias fases:

- Escritura del código: El programador escribe el código fuente en un lenguaje de programación.
- Compilación: El compilador convierte el código fuente en código objeto (lenguaje de máquina) específico para el hardware.
- Ensamblaje: El código objeto se convierte en un formato que el procesador puede entender.
- Enlazado: Las diferentes partes del programa, así como las bibliotecas externas, se combinan para formar el ejecutable final.
- Importancia del Compilador y Enlazador: El compilador es clave para traducir el código a un lenguaje que la computadora entienda, mientras que el enlazador se encarga de juntar todas las partes del programa, incluyendo las funciones y bibliotecas, para generar el archivo ejecutable.

1.5. Lógica y pensamiento computacional

La **lógica** es la base del pensamiento computacional, y se refiere a la capacidad de estructurar soluciones a problemas complejos de manera clara y secuencial. En la programación, esto significa descomponer un problema grande en sub-problemas más pequeños, resolverlos uno por uno y combinarlos para alcanzar la solución final.

- Descomposición de Problemas: En la programación, los problemas grandes se resuelven dividiéndolos en partes manejables, lo que facilita la creación de soluciones eficientes. Este proceso, llamado descomposición, es esencial para escribir programas bien estructurados.
- Importancia de la Lógica: La lógica es fundamental en la programación, ya que permite tomar decisiones y controlar el flujo del programa mediante estructuras condicionales y ciclos.

Ejemplo de Lógica y Pensamiento Computacional: Preparar un Sándwich

Problema: Quieres hacerte un sándwich, pero necesitas asegurarte de que tienes todos los ingredientes necesarios y seguir una secuencia lógica de pasos.

- Descomposición del Problema:
  - Paso 1: Verificar si tienes pan.
  - Paso 2: Verificar si tienes los ingredientes para el relleno (queso, jamón, etc.).
  - Paso 3: Si tienes pan y relleno, entonces puedes hacer el sándwich.
  - Paso 4: Si falta algún ingrediente, decide si irás a comprarlo o hacer otra cosa para comer.
- Decisiones Lógicas (Condicionales):
  - Condicional 1: ¿Tienes pan?
    - Si NO, decide si ir a comprar pan o hacer otra cosa para comer.
    - Si SÍ, continúa al siguiente paso.
  - Condicional 2: ¿Tienes los ingredientes del relleno?
    - Si NO, decide si comprar los ingredientes faltantes o hacer un sándwich con lo que tengas.
    - Si SÍ, procede a hacer el sándwich.

Este ejemplo muestra cómo la lógica y el pensamiento computacional te ayudan a descomponer el problema en partes más pequeñas (pan, ingredientes) y tomar decisiones basadas en las condiciones (si tienes o no los ingredientes necesarios).

1.6. Diagramas de flujo y pseudocódigo

Antes de escribir código, es útil planificar la lógica del programa:

- **Diagramas de flujo**: Representaciones gráficas que muestran el flujo de control de un algoritmo, utilizando símbolos estandarizados para diferentes tipos de acciones y decisiones.
- **Pseudocódigo**: Descripción textual de un algoritmo que combina elementos del lenguaje natural con estructuras de programación. No sigue la sintaxis de un lenguaje de programación específico, pero ayuda a visualizar la lógica antes de la implementación.

2. Paradigmas de programación

2.1. Introducción a los paradigmas de programación

Un **paradigma de programación** es un enfoque o estilo de escribir código que define cómo estructurar y organizar un programa para resolver problemas. A lo largo de la historia de la informática, han surgido distintos paradigmas con diferentes formas de pensar y modelar la solución de un problema en código.

Algunos paradigmas se enfocan en la manera en que se ejecutan las instrucciones (**imperativa vs. declarativa**), mientras que otros definen cómo se organiza la estructura del código (**estructurada, orientada a objetos, funcional, lógica, etc.**).

Un mismo lenguaje de programación puede admitir más de un paradigma. Por ejemplo, **Python** permite escribir código en un estilo **imperativo, estructurado, procedural, orientado a objetos y funcional**, dependiendo de cómo se use.

En este capítulo nos enfocaremos en los paradigmas más utilizados, con especial énfasis en la **programación estructurada**

2.1.2. Programación Imperativa

La **programación imperativa** es el paradigma más antiguo y se basa en dar instrucciones secuenciales al computador para modificar el estado del programa. Se le llama "imperativa" porque indica cómo deben ejecutarse las operaciones paso a paso.

Características principales:

- Uso de variables que almacenan valores y pueden modificarse.
- Uso de estructuras de control (**if**, **while**, **for**, etc.) para controlar el flujo de ejecución.
- El código se ejecuta de forma secuencial, de arriba hacia abajo.

2.1.3. Programación Estructurada (subtipo de imperativa)

La **programación estructurada** surgió en los años 70 como una evolución de la programación imperativa para mejorar la claridad y mantenimiento del código.

Principios básicos:

1. **Secuencia:** Instrucciones ejecutadas en orden.
2. **Selección:** Permite tomar decisiones (**if, if-else**).
3. **Iteración:** Repite un bloque de código (**while, for**).

Se eliminó el uso del **goto**, ya que producía código difícil de leer y mantener ("código espagueti").

En el caso de Python, **no admite goto de forma nativa**. Sin embargo, existe un módulo llamado **goto** en la librería **goto-statement** que permite simularlo en Python, aunque **su uso es altamente desaconsejado, tanto en Python como en cualquier otro lenguaje,** porque va en contra de las buenas prácticas de programación estructurada.

2.1.4. Programación Procedural (subtipo de estructurada)

También conocida como **programación basada en procedimientos**, es una extensión de la programación estructurada donde el código se organiza en **funciones** o **procedimientos** reutilizables.

Características:

- Divide el código en funciones que encapsulan bloques de lógica.
- Permite reutilización y mejor mantenimiento.
- Usa llamadas a funciones para ejecutar procedimientos específicos.

2.1.5. Programación Orientada a Objetos (POO)

La **POO** es un paradigma que modela los problemas en términos de **objetos** y **clases**. Es útil para desarrollar software modular y reutilizable.

Principios de la POO:

1. **Encapsulamiento:** Agrupa datos y funciones dentro de una clase.
2. **Herencia:** Permite que una clase reutilice y extienda el comportamiento de otra.
3. **Polimorfismo:** Permite usar el mismo método en diferentes contextos, entre otras cosas.

2.1.6. Programación Declarativa

A diferencia de la programación imperativa, que indica **cómo** hacer algo, la programación declarativa se enfoca en **qué** se quiere lograr, sin especificar paso a paso cómo se hace. Un ejemplo de este paradigma es **SQL**, que se usa para consultar bases de datos sin especificar cómo recuperarlas

3. Introducción a la programación con Python

Python es un lenguaje de programación de alto nivel, interpretado y multiparadigma, diseñado con un enfoque en la legibilidad y simplicidad del código. Su historia comienza a finales de la década de 1980, cuando **Guido van Rossum**, un programador neerlandés que trabajaba en el Centrum Wiskunde & Informatica (CWI) en los Países Bajos, decidió crear un lenguaje que combinara la potencia de otros lenguajes como ABC, pero con una sintaxis más clara y flexible.

Van Rossum quería un lenguaje fácil de aprender y usar, que permitiera a los programadores escribir código eficiente y elegante sin una sintaxis complicada. Durante la Navidad de 1989, comenzó a desarrollar Python en su tiempo libre, basándose en varios conceptos de ABC, pero con mejoras en la gestión de excepciones y la interacción con el sistema operativo.

El nombre **Python** no proviene de la serpiente, sino del grupo de comedia británico *Monty Python's Flying Circus*, un programa que Van Rossum disfrutaba. Desde el inicio, la filosofía de Python estuvo influenciada por el deseo de hacer que la programación fuera más accesible y divertida.

En **1991**, Python 1.0 fue lanzado oficialmente, incluyendo características como manejo de excepciones, funciones y módulos. Desde entonces, el lenguaje evolucionó constantemente, con la llegada de Python 2.0 en el año **2000**, que introdujo mejoras importantes como la recolección de basura y la compatibilidad con Unicode. Sin embargo, Python 2 tenía algunas limitaciones que llevaron a la creación de una versión completamente renovada.

En **2008**, se lanzó **Python 3.0**, una versión que introdujo cambios significativos en la sintaxis y la gestión de cadenas de texto, lo que rompió la compatibilidad con Python 2. Aunque la transición fue lenta, con muchos proyectos aún utilizando Python 2 durante varios años, en **2020** se dio oficialmente el fin del soporte para Python 2, consolidando a Python 3 como el estándar actual.

Hoy en día, Python es uno de los lenguajes de programación más utilizados en el mundo, con aplicaciones en desarrollo web, ciencia de datos, inteligencia artificial, automatización y muchas otras áreas. Su comunidad activa y la filosofía de código abierto han permitido que el lenguaje siga creciendo y evolucionando con cada nueva versión.

3.1. Concepto de Variable y Constante

3.1.1. ¿Qué es una Variable?

Una **variable** es un espacio en la memoria que almacena un valor y puede cambiar durante la ejecución del programa. En Python, una variable se define simplemente asignándole un valor, sin necesidad de declarar su tipo de dato.

```python
nombre = "Leopoldo"
edad = 29
altura = 1.73
es_estudiante = True
```

3.1.2. ¿Qué es una Constante?

Una **constante** es un valor que no cambia durante la ejecución del programa. En Python, no hay una forma nativa de definir constantes, pero por convención se escriben en **mayúsculas** para indicar que su valor no debería modificarse.

```
PI = 3.1416
GRAVEDAD = 9.81
```

Nota: Aunque Python permite modificar el valor de una "constante", se recomienda no hacerlo por buenas prácticas de programación.

3.2. Tipos de Datos Primitivos en Python

En programación, un **tipo de dato primitivo** es la unidad más básica de almacenamiento de información que maneja un lenguaje. Son la base para definir estructuras de datos más complejas y permiten representar valores esenciales como números, texto y valores lógicos.

Tipo de dato	Descripción	Ejemplo
int	Números enteros	edad = 29
float	Números decimales	altura = 1.73
str	Cadenas de texto	nombre = “Leopoldo”
bool	Valores lógicos	es_estudiante = True

"Un dato de tipo **bool** solo puede tener dos valores: **True** (verdadero) o **False** (falso). En Python, los valores booleanos también se pueden tratar como números enteros, donde True equivale a 1 y False equivale a 0."

3.3. Conversión de Tipos de Datos

En Python, la conversión de tipos de datos se conoce como **casting** y permite transformar un valor de un tipo a otro. Esto es útil cuando necesitamos operar con diferentes tipos de datos en un mismo programa.

Existen dos tipos de conversión:

3.3.1. Conversión Implícita (Automática)

Python realiza conversiones de manera automática cuando no hay riesgo de pérdida de datos. Ocurre principalmente entre tipos numéricos.

```
entero = 5        # int
decimal = 2.5     # float
resultado = entero + decimal
```

Python convierte entero en float antes de sumarlo con decimal, evitando pérdida de precisión.

3.3.2. Conversión Explícita (Casting)

Cuando Python no realiza la conversión automáticamente, debemos hacerla manualmente utilizando funciones de conversión.

Función	Conversión a
int(x)	Entero (trunca los decimales si es float)
float(x)	Número a decimal
str(x)	Cadena de texto
bool(x)	Booleano (False si el valor es 0, None o "")

```
num_float = 4.8
num_int = int(num_float)
print(num_int)

num = 10
cadena = str(num)
print(cadena)

texto = "3.14"
num_decimal = float(texto)
print(num_decimal)


print(bool(0))
print(bool(1))
print(bool(""))
print(bool("Hola"))
```

Resultado por pantalla

```
>>> %Run -c $EDITOR_CONTENT
 4
 10
 3.14
 False
 True
 False
 True
>>>
```

Cuidado con la conversión de strings a números

Si intentamos convertir una cadena no numérica en un número, obtendremos un error:

```
valor = int("Leopoldo")
```

3.4. Entrada y Salida de Datos en Línea de Comandos

3.4.1. Salida de Datos (print())

La función print() se usa para mostrar información en pantalla.

```
nombre = "Leopoldo"
edad = 29
print("Nombre:", nombre, "- Edad:", edad)
```

Salida por pantalla

```
>>> %Run -c $EDITOR_CONTENT
 Nombre: Leopoldo - Edad: 29
>>>
```

También podemos usar **f-strings** para formatear la salida de forma más clara:

```
nombre = "Carlos"
edad = 30
print(f"Nombre: {nombre} - Edad: {edad}")
```

Siendo la salida por pantalla igual a la anterior.
Para usar un f-string, simplemente se antepone la letra `f` o `F` a la cadena de texto y se colocan las variables o expresiones dentro de llaves `{}`. A lo largo del curso veremos más utilidades.

3.4.2. Entrada de Datos (`input()`)

La función input() permite al usuario ingresar datos por teclado.

```
nombre = input("Ingrese su nombre: ")
print("Hola, " + nombre + "!")
```

Salida por pantalla

```
>>> %Run -c $EDITOR_CONTENT
 Ingrese su nombre:
```

```
>>> %Run -c $EDITOR_CONTENT
 Ingrese su nombre: Leopoldo
 Hola, Leopoldo!
>>>
```

Nota: input() siempre devuelve un valor de tipo **string**, por lo que si necesitamos un número, debemos convertirlo.

Mediante la conversión explícita (casting), podemos asegurarnos de que los datos ingresados por el usuario sean del tipo adecuado para su procesamiento:

```
edad = int(input("Ingrese su edad: "))
altura = float(input("Ingrese su altura en metros: "))

print("Edad:", edad)
print("Altura:", altura)
```

```
>>> %Run -c $EDITOR_CONTENT
 Ingrese su edad: 29
 Ingrese su altura en metros:
```

```
>>> %Run -c $EDITOR_CONTENT
 Ingrese su edad: 29
 Ingrese su altura en metros: 1.73
 Edad: 29
 Altura: 1.73
>>>
```

4. Operadores Aritméticos y de Incremento

4.1. Operadores Aritméticos

Los **operadores aritméticos** permiten realizar operaciones matemáticas entre valores numéricos. En Python, los operadores aritméticos básicos son los siguientes:

Operador	Descripción	Ejemplo	Resultado
+	Suma	5 + 3	8
-	Resta	10 - 4	6
*	Multiplicación	7 * 2	14
/	División (devuelve float)	10 / 3	3.3333
//	División entera	10 // 3	3
%	Módulo (resto de la división)	10 % 3	1
**	Potencia	2 ** 3	8

```
x = 4
y = 5
z = 8.2

print(f"Suma: {x + y}")
print(f"Resta: {y - x}")
print(f"División entera: {z // x}")
print(f"División: {z / x}")
print(f"Multiplicación: {x * y}")
print(f"Potencia: {x ** y}")
print(f"Módulo: {y % x}")
```

```
>>> %Run -c $EDITOR_CONTENT
  Suma: 9
  Resta: 1
  División entera: 2.0
  División: 2.05
  Multiplicación: 20
  Potencia: 1024
  Módulo: 1

>>>
```

4.1.1. Inclusión de expresiones dentro de f-strings

Los **f-strings** en Python permiten no solo insertar variables dentro de una cadena, sino también realizar cálculos directamente dentro de las llaves **{}** sin necesidad de calcular previamente el resultado en una variable separada. Logrando así un código más limpio y conciso evitando crear variables intermedias para cálculos simples.

4.1.2. Operador Módulo (%) en Python

El **operador módulo (%)** devuelve el **resto** de la división entre dos números. Su uso es muy común en programación para determinar si un número es **par o impar**, encontrar múltiplos, realizar ciclos repetitivos, obtener los dígitos necesarios de un número, etc.

4.2. Operadores de Asignación

Los operadores de **asignación** en Python permiten almacenar valores en variables y, en algunos casos, modificar su contenido de manera simplificada. Se utilizan para asignar un valor a una variable y pueden combinarse con operadores aritméticos para realizar cálculos de forma más concisa.

4.2.1. Operador de asignación básica ( )

El operador = asigna un valor a una variable, como vimos en los ejemplos anteriores.

4.2.2. Operadores de asignación compuesta

Python ofrece operadores de asignación combinados con operadores aritméticos, lo que permite modificar el valor de una variable sin necesidad de repetir su nombre.

Operador	Descripción	Ejemplo equivalente
+=	Suma y asignación	x += 5 — x = x + 5
-=	Resta y asignación	x -= 5 — x = x - 5
*=	Multiplicación y asignación	x *= 5 — x = x * 5
/=	División y asignación	x /= 5 — x = x / 5
//=	División entera y asignación	x //= 5 — x = x // 5
%=	Módulo y asignación	x %= 5 — x = x % 5
**=	Potencia y asignación	x **= 5 — x = x ** 5

```
x = 10
print(x)
x += 5
print(x)
x *= 2
print(x)
x -= 10
print(x)
x /= 2
print(x)
```

Salida por pantalla

```
>>> %Run -c $EDITOR_CONTENT
 10
 15
 30
 20
 10.0

>>>
```

5. Estructuras condicionales

En programación, una **estructura condicional** permite ejecutar diferentes bloques de código según si una condición se cumple o no. Esto es fundamental para la toma de decisiones en un programa.

Python usa la palabra clave **if** para definir una condición, y opcionalmente se pueden agregar **else** y **elif** para manejar distintos casos. Por medio de ellas es posible que el programa defina que bloques de código ejecutar primero o directamente no hacerlo. Para ello es necesario también entender sobre la indentación, los operadores relacionales y los operadores lógicos.

5.1. Indentación

Python **no usa llaves {} ni palabras clave como begin y end** para delimitar bloques de código, a diferencia de lenguajes como C, Java o Pascal. En su lugar, **usa la indentación**, es decir, la sangría o el desplazamiento hacia la derecha en cada línea de código que pertenece a un bloque.
Cada vez que una línea de código termina con `:` (dos puntos), Python espera que el siguiente bloque de código esté **indentado**.

```python
edad = 20

if edad >= 18:
    print("Eres mayor de edad")
    print("Puedes votar")

print("Fin del programa")
```

5.1.1. Algunas reglas de Indentación

- **La indentación es obligatoria:** Si no indentas correctamente, Python mostrará un error de sangría (**IndentationError**).
- **Todos los elementos de un bloque deben tener la misma indentación:** Si mezclamos espacios y tabulaciones o usamos niveles distintos en un mismo bloque, Python lo detectará como un error.
- **Se recomienda usar 4 espacios por nivel de indentación:** La convención en Python (según PEP 8) es usar **4 espacios** en cada nivel.

5.2. Operadores Relacionales

Los **operadores relacionales** en Python se utilizan para comparar dos valores y devuelven un resultado de tipo **booleano (True o False)**. Son fundamentales para estructuras de control como **condicionales (if)**

Operador	Descripción	Ejemplo	Resultado
==	Igual a	5 == 5	True
!=	Distinto de	5 != 3	True
>	Mayor a	5 > 8	False

<	Menor a	3 < 1	False
>=	Mayor o igual a	5 >= 4	True
<=	Menor o igual a	4 <= 2	False

Como vimos en el ejemplo, si la edad es 18 o más, la persona es mayor de edad y por lo tanto puede votar.

5.3. Operadores lógicos

En programación, muchas veces necesitamos evaluar **más de una condición** dentro de una misma estructura condicional. Para esto, utilizamos los **operadores lógicos**, que nos permiten **combinar múltiples condiciones en una sola expresión lógica**.

Estos operadores toman como entrada valores **booleanos** (**True** o **False**) y devuelven como resultado otro valor booleano.

Operador	Descripción	Ejemplo
and (Y)	Devuelve True si **ambas condiciones** son verdaderas	(edad > 18 and registrado)
or (O)	Devuelve True si **al menos una** de las condiciones es verdadera	(usuario_es_admin or tiene_permisos)
not (NO)	Invierte el valor de una condición (True → False y viceversa)	not es_suscriptor

Si queremos, por ejemplo, determinar si un usuario puede acceder a un sistema:

- Debe tener más de **18 años**
- Debe estar **registrado**

Sin operadores lógicos, tendríamos que hacer algo como esto:

```
edad = 20
registrado = True

if edad > 18:
    if registrado:
        print("Acceso permitido")
```

En cambio, usando operadores lógicos, podemos escribirlo de manera más sencilla:

```
if edad > 18 and registrado:
    print("Acceso permitido")
```

5.4. Estructuras

5.4.1. if

Tal como vimos en los ejemplos anteriores, la estructura if evalúa una condición y, si es True, ejecuta el bloque de código correspondiente.

5.4.2. if con else, dos caminos posibles

A veces, necesitamos que una acción se ejecute si la condición es **False**. Para esto, usamos **else**.

```
edad = int(input("Ingrese su edad: "))

if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

En este caso, se cumpla o no la condición, el programa notifica al usuario, ya sea por el lado verdadero como por el lado falso.

```
>>> %Run -c $EDITOR_CONTENT
 Ingrese su edad: 15
 Eres menor de edad

>>>
```

5.4.3. if, elif y else, múltiples condiciones y caminos posibles

Cuando hay **más de dos posibles escenarios**, usamos **elif** (abreviatura de *else if*).

```
nota = 85

if nota >= 90:
    print("Calificación: A")
elif nota >= 80:
    print("Calificación: B")
elif nota >= 70:
    print("Calificación: C")
else:
    print("Calificación: Reprobado")
```

Importante:

- **Solo una condición se ejecuta**, la primera que sea **True**.
- Si ninguna se cumple, Python ejecuta el bloque **else**.
- **Podemos combinar condiciones** con operadores lógicos.
- Si es necesario se pueden anidar condiciones cuando son expresiones muy complejas y sea de difícil lectura en una sola confición.

5.4.4. match

A partir de Python 3.10, podemos usar **match-case**, que se asemeja a **switch** de otros lenguajes. Esta opción puede ser más acorde cuando hay múltiples caminos logrando mayor legibilidad.

```python
print("Opciones")
print("1. opción 1")
print("2. opción 2")
print("3. opción 3")
print("4. opción 4")
opcion = int(input("Elija una opción"))


match opcion:
    case 1:
        print("Elegiste la opción 1")
    case 2:
        print("Elegiste la opción 2")
    case 3:
        print("Elegiste la opción 3")
    case 4:
        print("Elegiste la opción 4")
    case _:
        print("Opción no válida")
```

Explicación:

- **match opcion:** evalúa el valor de opcion.
- Cada **case** equivale a un **case en switch** de de otros lenguajes.
- _ (guión bajo) actúa como **default**, cubriendo todas las opciones no listadas.

5.5. Expresiones

Una expresión en Python puede ser tan simple como un valor literal o una variable, o tan compleja como una combinación de múltiples operadores y funciones.

5.5.1. Ejemplos de expresiones simples

```python
x = 10
nombre = "Leopoldo"
es_estudiante = True
```

5.5.2. Ejemplos de expresiones compuestas

```python
x = 10 + 5 * 2
resultado = (x > 10) and (x < 30)
mensaje = f"El resultado es {x}"
```

5.5.1. Precedencia y Asociatividad de Operadores

Orden

Precedencia	Operadores	Descripción
1	()	Paréntesis, se evalúan primero
2	**	Potencias
3	*, /, //, %	Multiplicación, divisiones y módulo
4	+, -	Sumas y restas
5	==, !=, >, >=, <, <=	Operadores relacionales
6	not	Negación lógica
7	and	Operador lógico AND
8	or	Operador lógico OR
9	=, +=, -=, *=, /=, //=, %=, **=	Operadores de asignación

```
x = 10 + 5 * 2  # El resultado es 20
x = (10 + 5) * 2 # El resultado es 30
x = 10 + 5**2 # El resultado es 35
x = 8
x += x *6 # El resultado es 245
```

Si dos operadores tienen la misma precedencia, se evalúan según su **asociatividad**:

- **Asociatividad izquierda**: Se evalúa de izquierda a derecha (la mayoría de los operadores).
- **Asociatividad derecha**: Se evalúa de derecha a izquierda (como ** y =).

Evaluación en Expresiones Booleanas

Las expresiones booleanas siguen una evaluación perezosa (**short-circuit evaluation**). Esto significa que Python **deja de evaluar** una expresión lógica tan pronto como el resultado es determinante.

```python
print("Números pares O mayores de 16")
numero = int(input("Ingrese un numero: "))

if numero > 16 or numero % 2 == 0:
   print(f"Ingresó: {numero}... El número entra")
else:
    print(f"Ingresó: {numero}... El número no entra")
print("------------------------------")
print("Números pares Y mayores de 16")
numero = int(input("Ingrese un numero: "))

if numero > 16 and numero % 2 == 0:
   print(f"Ingresó: {numero}... El número entra")
else:
   print(f"Ingresó: {numero}... El número no entra")
```

En el primer ejemplo, si el número es mayor a 16, cumpliéndose la primera condición: numero > 16 ya no evaluará la segunda que es. numero % 2 == 0 porque una de las dos se cumple y así actúa OR para que sea True.
Por el contrario, en el segundo ejemplo, ambas condiciones deben cumplirse para ser True.

Tabla de verdad

A	B	A and B	A or B	Not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

- **and (Y lógico):** Sólo es True si **ambas condiciones** son verdaderas.
- **or (O lógico):** Es True si **al menos una** de las condiciones es verdadera.
- **not (Negación lógica):** Invierte el valor lógico: **True** → **False** y **False** → **True**.

6. Estructuras de repetición

En programación, la **iteración** es el proceso de repetir una serie de instrucciones múltiples veces hasta que se cumpla una condición específica. Esta característica es fundamental porque permite automatizar tareas repetitivas, optimizar el código y reducir la cantidad de líneas necesarias para resolver problemas.

Existen dos tipos principales de iteración:

- **Iteración con un número determinado de repeticiones**: Se usa cuando sabemos de antemano cuántas veces queremos repetir un bloque de código. En Python, esto se logra principalmente con el bucle **for.**
- **Iteración con una condición de control**: En este caso, la repetición continúa mientras se cumpla una condición lógica. En Python, este tipo de iteración se implementa con el bucle **while.**

Además, el uso de iteraciones permite evitar código redundante y mejorar la eficiencia de los programas.

6.1. Ciclo while

El bucle **while** en Python permite ejecutar un bloque de código **mientras se cumpla una condición lógica**. Esto significa que la cantidad de iteraciones no está definida de antemano, sino que depende de la evaluación de una condición booleana.

```
contador = 1
while contador <= 5:
    print("vuelta número", contador)
    contador += 1

print("Fin del programa")
```

1. La variable **contador** inicia en 1.
2. Mientras **contador** sea menor o igual a 5, se ejecuta el **print().**
3. En cada iteración, se incrementa **contador.**
4. Cuando **contador** es 6, la condición **contador <= 5** es **falsa** y el bucle se detiene.

Salida por pantalla

```
>>> %Run -c $EDITOR_CONTENT
 vuelta número 1
 vuelta número 2
 vuelta número 3
 vuelta número 4
 vuelta número 5
 Fin del programa

>>>
```

6.1.1 ¡Cuidado con los bucles infinitos!

Si la condición del **while nunca se vuelve falsa**, el bucle se ejecutará indefinidamente, causando que el programa no termine.
Por ejemplo, el siguiente código nunca se detiene porque **contador** nunca cambia:

```
contador = 1
while contador <= 5:
    print("Esto nunca termina")

print("Fin del programa")
```

6.1.2. Ciclo while con Entrada del Usuario

Este programa pide números positivos y los suma, cortando el ciclo cuando se ingresa un número negativo, al final muestra el resultado.

```
suma = 0
numero = int(input("Ingrese un número positivo (o negativo para salir): "))

while numero >= 0:
    suma += numero
    numero = int(input("Ingrese otro número positivo: "))

print("La suma total es:", suma)
```

Como observamos, hay una instrucción para ingreso de datos, habilita al ciclo para que empiece a iterar y antes de que el ciclo termine la vuelta vuelve a repetir la instrucción, si esta última no estuviera sería imposible que el ciclo siga iterando ya que el programa no vuelve a ejecutar la institución que dio inicio.

6.2. Ciclo for

El bucle **for** permite repetir una acción un número determinado de veces. A diferencia de **while**, aquí sabemos de antemano cuántas veces se repetirá el código.

```
for variable in range(valor_inicial, valor_final):
    #línea de código
    #línea de código
    #línea de código
```

- **valor_inicial** es el número desde donde empieza.
- **valor_final** es el número hasta donde termina (**no se incluye**).

Salvo que tengamos un programa específico, se recomienda iniciar el ciclo en 0 ya que en programación empezamos a contar desde el mismo.

```
for x in range(0, 5):
    print("Repetición número", x + 1)

print("Fin del programa")
```

Si comenzamos desde el 0, no es necesario especificarlo, Python si no encuentra ese valor toma que se empieza desde el 0:

```
for x in range(5):
    print("Repetición número", x + 1)

print("Fin del programa")
```

Si iniciamos con 0, nuestra variable de inicio marcaría que la primera vuelta es 0, por eso es importante que cuando se muestre le sumemos 1 como en el ejemplo.

```
>>> %Run -c $EDITOR_CONTENT
 Repetición número 1
 Repetición número 2
 Repetición número 3
 Repetición número 4
 Repetición número 5
 Fin del programa

>>>
```

6.2.1. Uso de break para detener ciclo

Si queremos **detener** el **for** antes de que termine, usamos **break**.

```
for x in range(10):
    if x == 5:
        break
    print("Numero de vuelta", x + 1)

print("Fin del programa")
```

De esta forma, con la estructura condicional que tenemos dentro del ciclo for se evalúa que número de vuelta es, cuando llegue x sea igual a 5 finaliza el ciclo.

```
>>> %Run -c $EDITOR_CONTENT
 Numero de vuelta 1
 Numero de vuelta 2
 Numero de vuelta 3
 Numero de vuelta 4
 Numero de vuelta 5
 Fin del programa
>>>
```

Recordar que cuando le sumamos 1 a x no alteramos su valor, solo se muestra incrementado pero su valor real sigue siendo 1 unidad menos.

6.2.2. Uso de continue para saltar una vuelta

Si queremos **saltar** una repetición sin detener el ciclo, usamos **continue.**

```
for x in range(6):
    if x == 2:
        continue
    elif x == 4:
        continue
    print("Vuelta número", x + 1)
```

En este caso, si x vale 2 o 4 las vueltas no se mostrarán por pantalla, pero al sumar 1 unidad a x al mostrarse serán las vueltas 3 y 5:

```
>>> %Run -c $EDITOR_CONTENT
 Vuelta número 1
 Vuelta número 2
 Vuelta número 4
 Vuelta número 6
>>>
```

6.3. Ciclos combinados

Son estructuras en las que un ciclo está **dentro** de otro (anidamiento de bucles) o donde se combinan diferentes tipos de ciclos (**for** con **while,** etc.).

```
for x in range(3):
    for y in range(2):
        print(f"x = {x}, y = {y}")
```

Cada vuelta de x contiene dentro 2 vueltas de y. De esta forma, cada vuelta del ciclo de x tiene un ciclo de vueltas de y.

```
>>> %Run -c $EDITOR_CONTENT
 x = 0, y = 0
 x = 0, y = 1
 x = 1, y = 0
 x = 1, y = 1
 x = 2, y = 0
 x = 2, y = 1

>>>
```

6.3.1 Ejemplo más complejo

Se necesita un programa para cargar las ventas ordenadas por mes del año 2024. El ingreso de ventas cambia de mes cuando se ingresa 0.La cantidad de ingresos de ventas no son las mismas en cada mes. Al final del programa se debe mostrar el total de las ventas de todos los meses

Análisis del enunciado

ordenadas por mes del año 2024: sabemos que todos los años tienen la misma cantidad de meses por lo que sabemos de antemano la cantidad de vueltas que dará el ciclo externo (12), ciclo for.
La cantidad de ingresos de ventas no son las mismas en cada mes: No sabemos cuántas ventas hubo por mes, entonces la cantidad de vueltas que de el ciclo interno dependerá del usuario y la cantidad de ventas que deba cargar, ciclo while.
El ingreso de ventas cambia de mes cuando se ingresa 0: El fin de carga de ventas por mes está regido por el 0, es decir que cuando termino de cargar ventas de un mes debo ingresar 0 para que cambie el mes.
Al final del programa se debe mostrar el total de las ventas de todos los meses: Cuando todas las vueltas de ambos ciclos finalizan, el programa debe mostrar lo recaudado en total de todos los meses. Es decir que se necesita acumular todas las ventas.

```python
venta = 0
acumulador_ventas = 0

for x in range(12):
    print("Mes:", x + 1)
    venta = int(input("Ingrese importe de venta: "))
    while venta != 0:
        acumulador_ventas += venta
        venta = int(input("Ingrese importe de venta"))

print(f"El total de ventas en el año es: ${acumulador_ventas}")
```

- Se inician las variables de venta y acumulador_ventas en 0.
- Un ciclo for de 12 vueltas, uno por cada mes, indica en que mes estamos.
- Dentro del ciclo for un ciclo while que pide ingresar ventas.
- Una variable acumuladora se encarga de sumar en cada vuelta el importe de la venta que se ingresa.
- Al final del programa se muestra el total acumulado durante el año.