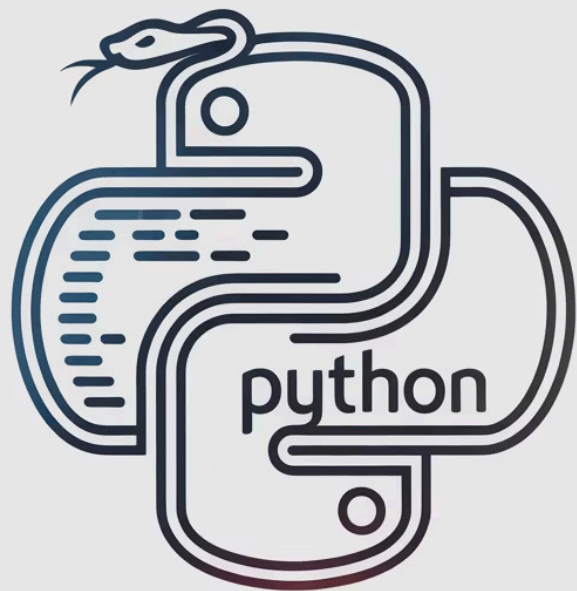


# Introducción al Backend y Arquitectura Cliente-Servidor

Curso: Desarrollo Backend con Django

Universidad de los Andes | Vigilada Mineducación. Reconocimiento como Universidad: Decreto 1297 del 30 de mayo de 1964.  
Reconocimiento personería jurídica: Resolución 28 del 23 de febrero de 1949 MinJusticia.



# Fundamentos de Python

# Agenda

1

## Sintaxis básica de Python

Variables, comentarios y operadores

2

## Tipos de datos

Primitivos y compuestos

3

## Funciones

Declaración y uso de funciones

4

## Estructuras de control

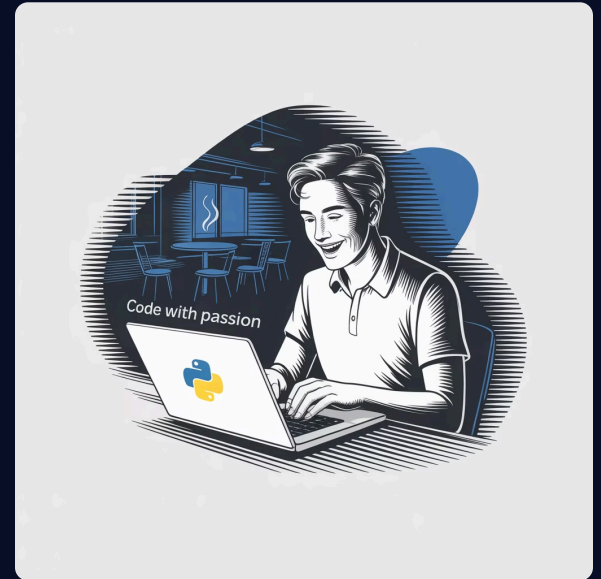
Condicionales y bucles

# ¿Por qué Python?

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, creado por Guido van Rossum y lanzado por primera vez en 1991. Se ha convertido en uno de los lenguajes más populares a nivel mundial gracias a su filosofía de diseño que enfatiza la legibilidad del código y su amplia aplicabilidad en diversos campos.

## Características clave y ventajas:

- **Fácil de aprender y usar**
- **Sintaxis limpia y legible**
- **Versatilidad increíble**
- **Gran comunidad de soporte**
- **¡Muy divertido de usar!**



# Sintaxis básica de Python

# Variables en Python

Las variables en Python son contenedores que almacenan valores de datos. No necesitan declaración de tipo explícita, ya que Python determina automáticamente el tipo basándose en el valor asignado.




Simples, flexibles y sin declaración de tipo

```
nombre = "Ana"  
edad = 25  
altura = 1.65  
es_estudiante = True
```



# Reglas para nombrar variables

Para escribir código Python claro, legible y fácil de mantener, es fundamental seguir ciertas convenciones al nombrar las variables. Estas reglas aseguran la consistencia y evitan errores comunes.

-  **Deben comenzar con una letra o \_**  
**Correcto:** nombre, \_valor  
**Incorrecto:** 1nombre, @valor
-  **Sensibles a mayúsculas/minúsculas**  
edad, Edad y EDAD son diferentes
-  **Usar snake\_case (recomendado)**  
nombre\_completo = "Juan Pérez"

# Comentarios en Python

Los comentarios son texto explicativo que se incluye en el código para hacerlo más comprensible. Son fundamentales para documentar el propósito y funcionamiento del código.

```
# Esto es un comentario de una línea
```

```
"""
```

```
Este es un comentario  
de múltiples líneas  
Útil para documentación  
"""
```

```
edad = 25 # Comentario al final de una línea
```

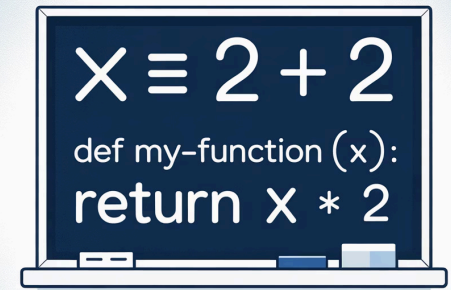
Los comentarios son ignorados por el intérprete de Python



# Operadores aritméticos

Los operadores aritméticos permiten realizar cálculos matemáticos básicos en Python. Son fundamentales para cualquier operación numérica.

+	Suma
-	Resta
*	Multiplicación
/	División
**	Exponente
//	División entera
%	Módulo (residuo)



“Code is Math”

# Operadores de comparación

Los operadores de comparación se utilizan para comparar dos valores. Devuelven un valor booleano: **True** si la condición es verdadera y **False** si es falsa.

==	Igual a	<code>x == y</code>
!=	Diferente de	<code>x != y</code>
>	Mayor que	<code>x &gt; y</code>
<	Menor que	<code>x &lt; y</code>
>=	Mayor o igual que	<code>x &gt;= y</code>
<=	Menor o igual que	<code>x &lt;= y</code>

Devuelven **True** o False

# Operadores lógicos

Los operadores lógicos se utilizan para combinar expresiones condicionales, permitiendo evaluar múltiples condiciones y tomar decisiones basadas en si todas, algunas o ninguna de ellas son verdaderas.

Son fundamentales para construir lógica compleja en programas y controlar el flujo de ejecución.

## **and**

Verdadero si ambas condiciones son verdaderas

```
x > 5 and x < 10
```

## **or**

Verdadero si al menos una condición es verdadera

```
x < 5 or x > 10
```

## **not**

Invierte el resultado

```
not(x > 5)
```

# Primer script en Python

Este script demuestra conceptos básicos de Python: importación de módulos, uso de variables, funciones integradas y formateo de strings. Es un ejemplo práctico que muestra información del sistema.

```
# mi_primer_script.py
import platform
import os

# Obtener información del sistema
sistema = platform.system()
version = platform.version()
directorio = os.getcwd()

# Mostrar información
print("¡Hola mundo desde Python!")
print(f"Sistema operativo: {sistema}")
print(f"Versión: {version}")
print(f"Directorio actual: {directorio}")
```

# Ejecutando nuestro script

Ejecutar un script de Python significa indicarle a tu sistema que procese y lleve a cabo las instrucciones que escribiste en el archivo. Esto te permite ver el resultado de tu programa y verificar su funcionamiento directamente desde la línea de comandos.

```
$ python mi_primer_script.py
```

```
¡Hola mundo desde Python!
```

```
Sistema operativo: Windows
```

```
Versión: 10.0.19044
```

```
Directorio actual: C:\Users\estudiante\python
```

Ejecuta este código y verás información de tu propio sistema



# Tipos de datos en Python

# Tipos de datos en Python

En Python, los tipos de datos son fundamentales para clasificar y organizar la información que se manipula. Determinan el tipo de valores que una variable puede almacenar y las operaciones que se pueden realizar con ellos, lo que es esencial para escribir código claro y eficiente.

## Tipos primitivos

- int (enteros)
- float (decimales)
- str (cadenas de texto)
- bool (booleanos)

## Tipos compuestos

- list (listas)
- tuple (tuplas)
- dict (diccionarios)
- set (conjuntos)

# Tipos de datos primitivos

Los tipos de datos primitivos son los bloques fundamentales con los que se construye la información en programación. Representan los valores más básicos e indivisibles que una variable puede almacenar, como números, texto o valores lógicos.

1

**int**

```
edad = 25
```

Números enteros sin decimales

1.0

**float**

```
altura = 1.75
```

Números con decimales

66

**str**

```
nombre = "María"
```

Textos entre comillas



**bool**

```
es_estudiante = True
```

True o False



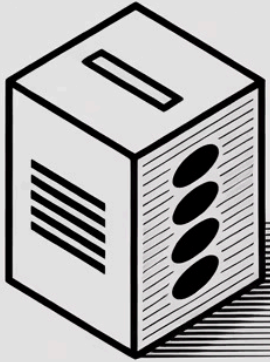
# Verificando tipos de datos

Para trabajar eficazmente con variables, es fundamental saber qué tipo de dato almacenan. La función `type()` es una herramienta incorporada en Python que nos permite determinar el tipo exacto de cualquier variable o valor. Esto es útil para depurar, validar entradas o realizar operaciones específicas según el tipo de dato.

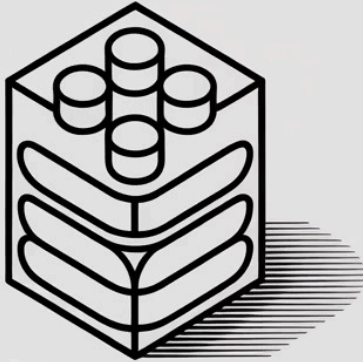
```
edad = 25
altura = 1.75
nombre = "Carlos"
es_estudiante = True

print(type(edad))      #
print(type(altura))    #
print(type(nombre))    #
print(type(es_estudiante)) #
```

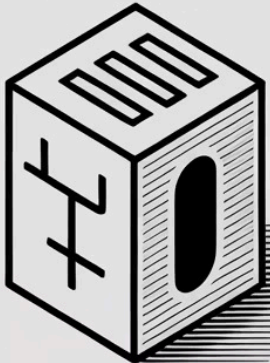
La función `type()` nos muestra el tipo de dato



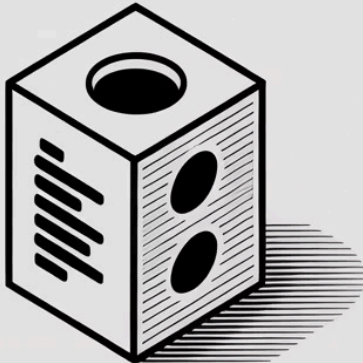
List



Dictionary



Tuple



Set

# Tipos de datos compuestos

Son estructuras que permiten almacenar y organizar múltiples valores dentro de una sola variable. Facilitan la manipulación de conjuntos de datos relacionados y cada tipo ofrece diferentes características para manejar la información de forma eficiente.

- **list:** Colección ordenada y modificable
- **tuple:** Colección ordenada e inmutable
- **dict:** Colección de pares clave-valor
- **set:** Colección desordenada sin duplicados

# Listas en Python

Las listas son colecciones ordenadas y modificables de elementos. Permiten almacenar múltiples valores en una sola variable y son muy versátiles para organizar datos.

# Creación de listas

```
frutas = ["manzana", "banano", "fresa"]
```

```
numeros = [1, 2, 3, 4, 5]
```

```
mixta = [1, "hola", True, 3.14]
```

# Acceder a elementos (índice desde 0)

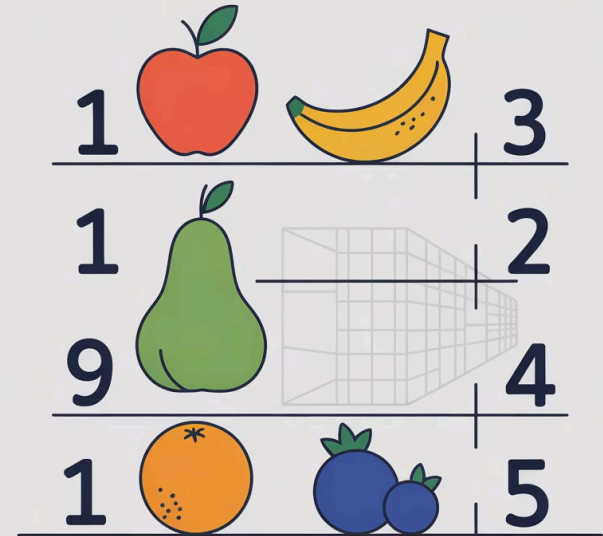
```
print(frutas[0]) # manzana
```

```
print(frutas[-1]) # fresa (último elemento)
```

# Modificar elementos

```
frutas[1] = "piña"
```

```
print(frutas) # ["manzana", "piña", "fresa"]
```



# Operaciones con listas

Las listas en Python son dinámicas, lo que significa que su contenido puede modificarse después de su creación. Estas operaciones permiten añadir, eliminar, organizar y acceder a información dentro de las listas de manera eficiente.

```
frutas = ["manzana", "banano", "fresa"]
```

```
# Añadir elementos
```

```
frutas.append("naranja") # ["manzana", "banano", "fresa", "naranja"]
```

```
frutas.insert(1, "piña") # ["manzana", "piña", "banano", "fresa", "naranja"]
```

```
# Eliminar elementos
```

```
frutas.remove("banano") # ["manzana", "piña", "fresa", "naranja"]
```

```
ultima = frutas.pop() # Elimina y retorna "naranja"
```

```
# Longitud de la lista
```

```
print(len(frutas)) # 3
```

```
# Ordenar lista
```

```
frutas.sort() # ["fresa", "manzana", "piña"]
```

# Tuplas en Python

Las tuplas son colecciones ordenadas e inmutables de elementos. A diferencia de las listas, una vez creadas, sus elementos no pueden ser modificados, añadidos o eliminados. Son útiles para almacenar datos relacionados que no deberían cambiar durante la ejecución del programa.



## Inmutables

No se pueden modificar después de crearse



## Más rápidas

Mayor rendimiento que las listas



## Seguras

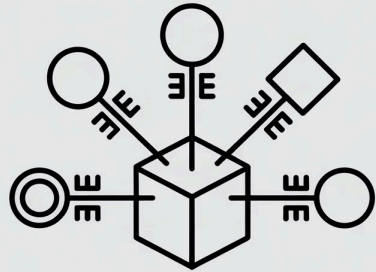
Protegen datos que no deben cambiar

```
coordenadas = (4.7110, -74.0721) # Bogotá  
dias = ("lunes", "martes", "miércoles", "jueves", "viernes")
```

```
# Acceder a elementos (igual que listas)  
print(coordenadas[0]) # 4.7110  
print(dias[1:3]) # ("martes", "miércoles")
```

# Diccionarios en Python

Los diccionarios son colecciones de datos que almacenan elementos en pares clave-valor. Son ideales para organizar información de manera lógica y acceder a ella rápidamente mediante sus claves únicas, similares a un diccionario de la vida real donde buscas por palabra (clave) para encontrar su definición (valor).



Knowledge unlocks

Almacenan datos en pares `clave-valor`

```
estudiante = {  
    "nombre": "Laura",  
    "edad": 22,  
    "carrera": "Ingeniería",  
    "activo": True  
}  
  
# Acceder a valores  
print(estudiante["nombre"]) # Laura  
  
# Modificar valores  
estudiante["edad"] = 23
```

# Operaciones con diccionarios

Los diccionarios en Python son dinámicos y permiten modificar su contenido. Puedes añadir, eliminar y verificar la existencia de elementos, así como acceder a sus claves y valores, lo que los hace muy flexibles para manejar colecciones de datos.

```
estudiante = {"nombre": "Laura", "edad": 22, "carrera": "Ingeniería"}

# Añadir nuevos pares clave-valor
estudiante["ciudad"] = "Medellín"

# Eliminar pares
del estudiante["edad"]

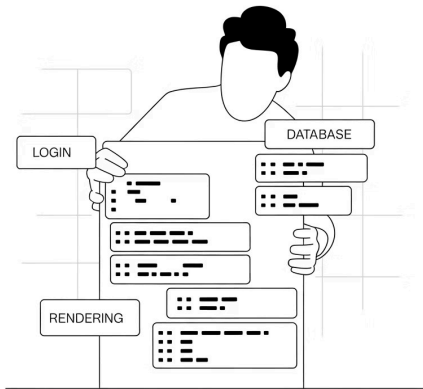
# Verificar si una clave existe
if "carrera" in estudiante:
    print("La clave carrera existe")

# Obtener todas las claves y valores
print(estudiante.keys()) # dict_keys(['nombre', 'carrera', 'ciudad'])
print(estudiante.values()) # dict_values(['Laura', 'Ingeniería', 'Medellín'])
print(estudiante.items()) # dict_items([('nombre', 'Laura'), ...])
```

# Funciones en Python



# ¿Qué son las funciones?



Las funciones son uno de los conceptos más importantes en programación. Permiten organizar y reutilizar código de manera eficiente.

Bloques de código reutilizables que:

- Realizan una tarea específica
- Pueden recibir datos (parámetros)
- Pueden devolver resultados
- Se definen una vez, se usan muchas veces

# Declaración de funciones

La declaración de una función es el proceso de definir su estructura y comportamiento. Es donde le damos un nombre, especificamos los datos que puede recibir (parámetros) y escribimos el código que ejecutará. Al declarar una función, creamos un bloque de código reutilizable listo para ser invocado cuando sea necesario.

En Python, usamos la palabra clave `def` seguida del nombre de la función y paréntesis para declararla. La indentación define qué código pertenece a la función.

```
# Estructura básica
def nombre_funcion(parametro1, parametro2):
    # Código de la función
    # ...
    return resultado # opcional

# Ejemplos
def saludar():
    print("¡Hola, bienvenido!")

def sumar(a, b):
    return a + b
```

# Llamando a funciones

Llamar a una función significa ejecutar el bloque de código que contiene. Una vez definida, puedes reutilizar la función tantas veces como necesites, pasando los argumentos requeridos para que realice su tarea.

```
# Definición de funciones
def saludar():
    print("¡Hola, bienvenido!")

def sumar(a, b):
    return a + b

# Llamada a funciones
saludar() # ¡Hola, bienvenido!

resultado = sumar(5, 3)
print(resultado) # 8

print(sumar(10, 20)) # 30
```

# Parámetros y argumentos

Los parámetros son nombres de variables definidos en la función para recibir valores, mientras que los argumentos son los valores reales que se envían a la función cuando se le llama. Son esenciales para hacer que las funciones sean flexibles y reutilizables, permitiendo que operen con diferentes datos.

## Parámetros

Variables que se definen en la función

```
def saludar(nombre):  
    print(f"Hola {nombre}")
```

## Argumentos

Valores que se pasan a la función

```
saludar("María") # Hola María  
saludar("Juan") # Hola Juan
```

# Tipos de parámetros

Python ofrece diversas formas de definir y pasar parámetros a las funciones, lo que proporciona flexibilidad al invocar el código. Comprender estos tipos es fundamental para escribir funciones claras y eficientes que puedan adaptarse a diferentes escenarios de uso.

## Parámetros posicionales

```
def describir(nombre, edad):  
    print(f"{nombre} tiene {edad} años")  
  
describir("Ana", 25) # Ana tiene 25 años
```

## Parámetros con valores por defecto

```
def saludar(nombre, mensaje="Hola"):   
    print(f"{mensaje}, {nombre}")  
  
saludar("Carlos") # Hola, Carlos  
saludar("Carlos", "Buenas tardes") #  
Buenas tardes, Carlos
```

## Parámetros con nombre

```
describir(edad=30, nombre="Luis") # Luis  
tiene 30 años
```

# Parámetros avanzados

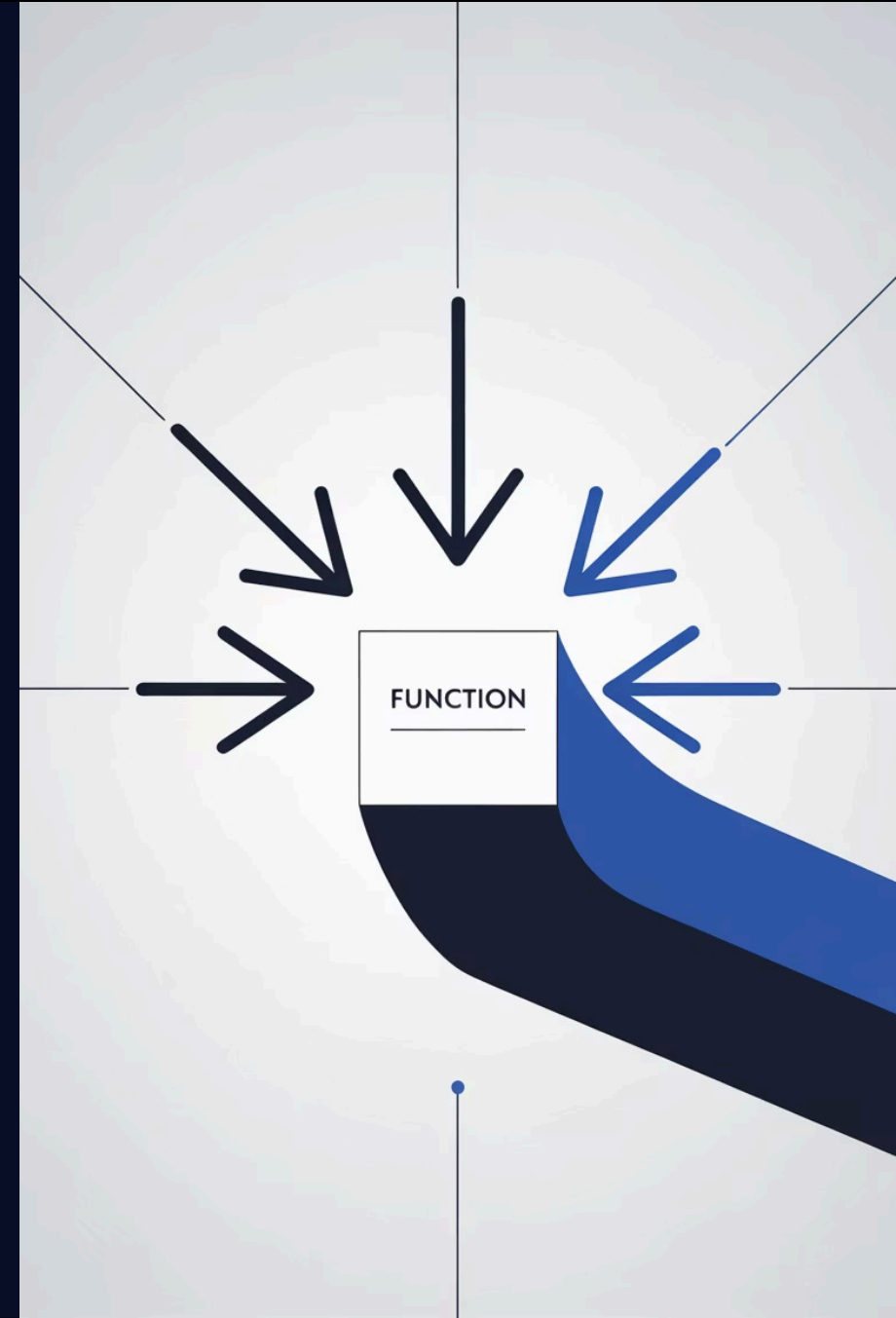
Los parámetros avanzados, como `*args` y `**kwargs`, permiten a las funciones aceptar un número variable de argumentos. Son útiles cuando no se sabe de antemano cuántos argumentos se pasarán a la función, proporcionando gran flexibilidad.

```
# *args: Número variable de argumentos posicionales
def sumar_todos(*numeros):
    return sum(numeros)

print(sumar_todos(1, 2, 3, 4)) # 10

# **kwargs: Número variable de argumentos con nombre
def info_persona(**datos):
    for clave, valor in datos.items():
        print(f'{clave}: {valor}')

info_persona(nombre="Ana", edad=25, ciudad="Cali")
# nombre: Ana
# edad: 25
# ciudad: Cali
```



# Retorno de valores

El retorno de valores en funciones permite que una función envíe uno o más resultados de vuelta al lugar desde donde fue llamada. Esto es fundamental para que las funciones realicen cálculos o procesen datos y luego pongan esos resultados a disposición de otras partes del programa.

```
# Retornar un valor
def calcular_iva(precio):
    return precio * 0.19

# Retornar múltiples valores
def dividir(a, b):
    cociente = a // b
    resto = a % b
    return cociente, resto

# Sin retorno explícito
def saludar(nombre):
    print(f"Hola {nombre}")
    # Retorna None implícitamente
```

```
# Usando los valores retornados
precio_final = calcular_iva(1000) + 1000
print(precio_final) # 1190

resultado = dividir(10, 3)
print(resultado) # (3, 1)

# Desempaquetar múltiples valores
c, r = dividir(10, 3)
print(f"Cociente: {c}, Resto: {r}")
# Cociente: 3, Resto: 1
```

# Alcance de variables

El alcance o ámbito de las variables define la visibilidad y accesibilidad de las variables en diferentes partes de un programa. Permite controlar dónde una variable puede ser referenciada y modificada, lo que es crucial para evitar conflictos de nombres y gestionar el flujo de datos.

## Variables locales

Solo existen dentro de la función

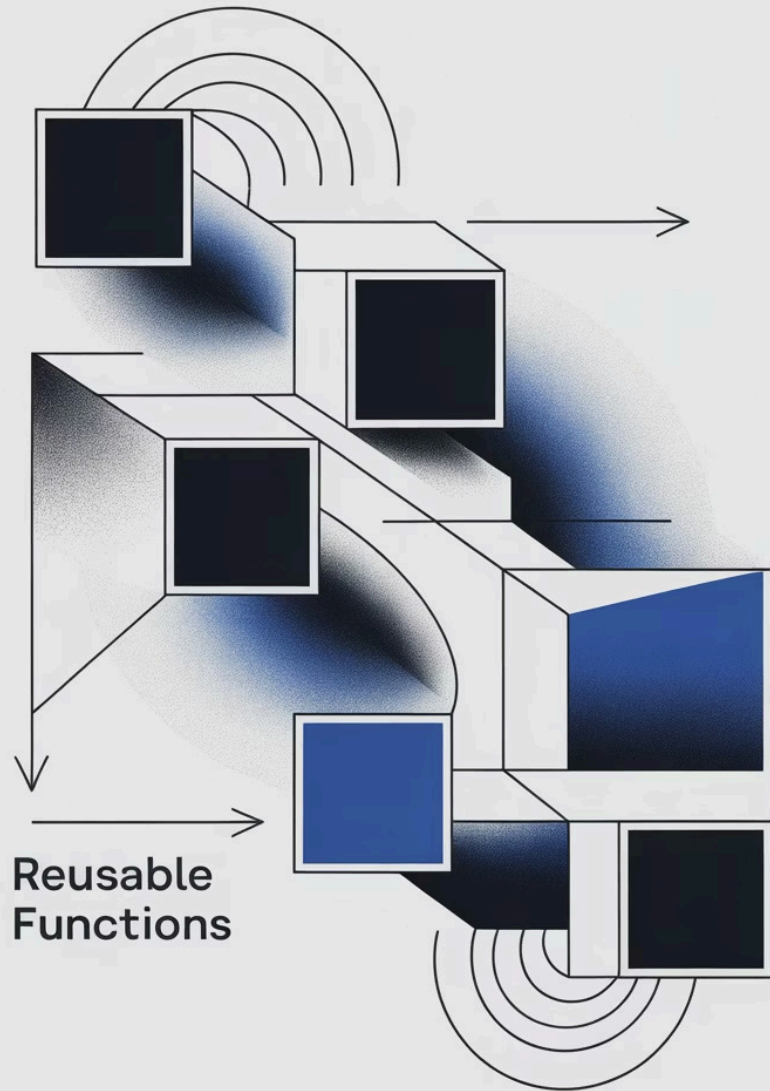
```
def mi_funcion():  
    x = 10 # Variable local  
    print(x)  
  
mi_funcion()  
# print(x) # ¡Error! x no existe fuera de la función
```

## Variables globales

Existen en todo el programa

```
y = 20 # Variable global  
  
def otra_funcion():  
    print(y) # Podemos usar la variable global  
    # Para modificarla dentro de la función:  
    global y  
    y = 30  
  
otra_funcion()  
print(y) # 30
```





# Ventajas de usar funciones

Las funciones son bloques de código reutilizables que agrupan un conjunto de instrucciones para realizar una tarea específica. Su uso permite modularizar el programa, haciendo el código más legible y manejable.



## Reutilización

Escribe una vez, usa muchas veces



## Organización

Código más estructurado



## Mantenimiento

Correcciones centralizadas



## Colaboración

Facilita el trabajo en equipo

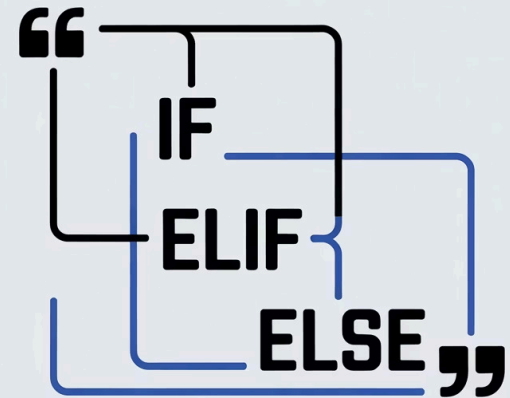
# Estructuras de control

# Estructuras condicionales

Las estructuras condicionales permiten que un programa ejecute diferentes bloques de código según si una o varias condiciones son verdaderas o falsas. Son fundamentales para la toma de decisiones dentro del flujo de un programa.

```
# Estructura básica
if condicion:
    # Código si la condición es True
elif otra_condicion:
    # Código si otra_condicion es True
else:
    # Código si ninguna condición es True

# Ejemplo
edad = 18
if edad < 18:
    print("Eres menor de edad")
elif edad == 18:
    print("Acabas de cumplir la mayoría de edad")
else:
    print("Eres mayor de edad")
```



# Operadores de comparación en condicionales

Los operadores de comparación se utilizan para comparar dos valores y devolver un resultado booleano (verdadero o falso). Son fundamentales para establecer las condiciones en las estructuras de control, como los enunciados `if`, `elif` y `else`, permitiendo que el programa tome decisiones basadas en estas comparaciones. Los operadores lógicos (`AND`, `OR`, `NOT`) se usan para combinar o modificar estas condiciones.

```
nota = 85

if nota >= 90:
    print("Excelente")
elif nota >= 80:
    print("Muy bien")
elif nota >= 70:
    print("Bien")
elif nota >= 60:
    print("Aprobado")
else:
    print("Reprobado")

# Operadores lógicos
edad = 25
tiene_carnet = True

if edad >= 18 and tiene_carnet:
    print("Puede conducir")
else:
    print("No puede conducir")
```

# Condicionales anidados

Los condicionales anidados son estructuras `if-else` colocadas dentro de otras. Permiten evaluar condiciones más específicas solo si una condición inicial se cumple, creando flujos de decisión complejos.

```
edad = 20
es_estudiante = True

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

    if es_estudiante:
        print("Tienes descuento de estudiante")
    else:
        print("No tienes descuento")
else:
    print("Eres menor de edad")
    print("Tienes descuento de menor")
```

¡Cuidado con anidar demasiados niveles! Reduce la legibilidad

# Operador ternario

El operador ternario es una forma concisa de escribir una sentencia condicional simple en una sola línea. Permite asignar un valor a una variable basado en el resultado de una condición, haciendo el código más legible y compacto para expresiones condicionales sencillas.

# Sintaxis

```
resultado = valor_si_verdadero if condicion else valor_si_falso
```

# Ejemplos

```
edad = 20
```

```
mensaje = "Mayor de edad" if edad >= 18 else "Menor de edad"
```

```
print(mensaje) # Mayor de edad
```

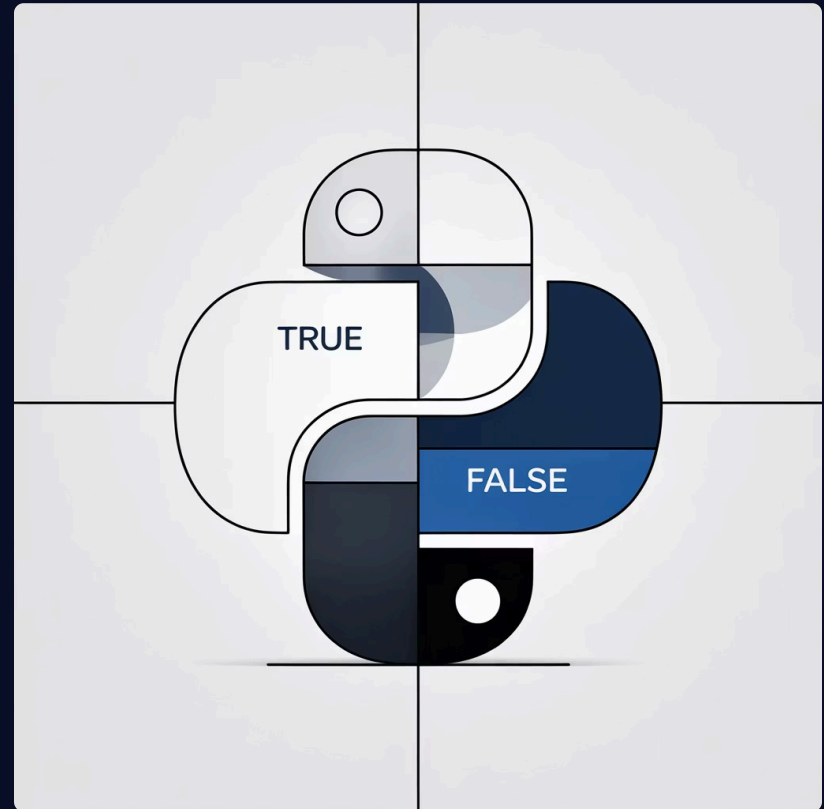
# Equivalente a:

```
if edad >= 18:
```

```
    mensaje = "Mayor de edad"
```

```
else:
```

```
    mensaje = "Menor de edad"
```



# Bucle for

Un bucle **for** se utiliza para iterar sobre una secuencia (como una lista, una tupla, un diccionario, un conjunto o una cadena) o sobre otros objetos iterables. Es útil para ejecutar un bloque de código repetidamente para cada elemento de la secuencia.

1

## Iterar sobre una secuencia

```
frutas = ["manzana", "fresa"]
for fruta in frutas:
    print(f"Me gusta la {fruta}")
```

```
# Me gusta la manzana
# Me gusta la fresa
```

2

## Iterar sobre un rango

```
for i in range(5): # 0, 1, 2, 3, 4
    print(i)
```

```
for i in range(2, 5): # 2, 3, 4
    print(i)
```

```
for i in range(0, 10, 2): # 0, 2, 4, 6, 8
    print(i)
```

3

## Iterar sobre diccionarios

```
estudiante = {"nombre": "Laura",
              "edad": 22}
for clave in estudiante:
    print(f"{clave}: {estudiante[clave]}")
```

```
# O mejor:
for clave, valor in estudiante.items():
    print(f"{clave}: {valor}")
```

# Bucle while

El bucle `while` es una estructura de control que permite ejecutar un bloque de código repetidamente mientras una condición específica sea verdadera. Se utiliza cuando el número de repeticiones no es conocido de antemano y depende de una condición dinámica.

```
# Estructura básica
while condicion:
    # Código a repetir
    # ...

# Ejemplo: contar hasta 5
contador = 1
while contador <= 5:
    print(contador)
    contador += 1

# Ejemplo: validación de entrada
respuesta = ""
while respuesta != "si":
    respuesta = input("¿Terminaste? (si/no): ")
```

¡Cuidado con los bucles infinitos!

```
# Bucle infinito
while True:
    print("Esto nunca termina")
    # Necesitamos un break
```



# Control de flujo en bucles

Las sentencias de control de flujo en bucles (`break`, `continue`, `else`) permiten modificar el comportamiento normal de la ejecución de un bucle, ofreciendo mayor flexibilidad y control sobre cómo se procesan las iteraciones.

## **break**

Termina el bucle inmediatamente

```
for i in range(10):  
    if i == 5:  
        break  
    print(i) # 0, 1, 2, 3, 4
```

## **continue**

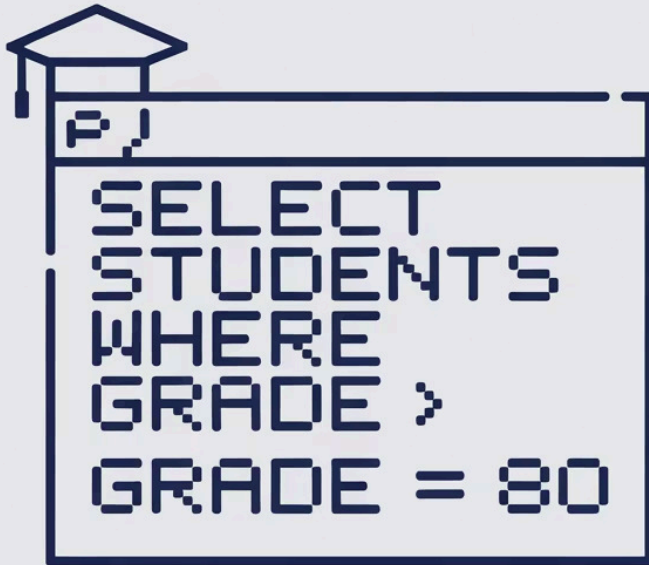
Salta a la siguiente iteración

```
for i in range(5):  
    if i == 2:  
        continue  
    print(i) # 0, 1, 3, 4
```

## **else**

Se ejecuta si el bucle termina normalmente (sin break)

```
for i in range(3):  
    print(i)  
else:  
    print("Bucle completado")
```



# Ejemplo práctico: Filtrar datos de estudiantes

La filtración de datos es una técnica fundamental en programación que permite seleccionar y extraer un subconjunto de información de una colección más grande, basándose en criterios específicos. Esto es crucial para analizar datos, generar informes o presentar solo la información relevante.

Vamos a crear un programa que filtre una lista de estudiantes y muestre solo aquellos que han aprobado.

Para este ejercicio, aplicaremos conceptos clave como: funciones, bucles, condicionales y diccionarios.

# Implementación del filtro de estudiantes

Ahora, prepararemos los datos iniciales para el filtro de estudiantes. Veremos la estructura de los datos que utilizaremos.

```
estudiantes = [  
    {"nombre": "Ana", "nota": 85},  
    {"nombre": "Carlos", "nota": 92},  
    {"nombre": "Elena", "nota": 78},  
    {"nombre": "David", "nota": 65},  
    {"nombre": "Laura", "nota": 90}  
]
```

Nuestro objetivo es crear una función que filtre esta lista y muestre solo los estudiantes que tienen una nota mayor o igual a 70 puntos.

# Función para filtrar estudiantes aprobados

Ahora, integramos los elementos que hemos visto (listas de diccionarios, bucles, condicionales) para construir la función completa que filtrará a nuestros estudiantes aprobados.

```
def filtrar_aprobados(lista, nota_minima=70):  
    aprobados = []  
    for estudiante in lista:  
        if estudiante["nota"] >= nota_minima:  
            aprobados.append(estudiante)  
    return aprobados
```

```
# Llamar a la función  
aprobados = filtrar_aprobados(estudiantes)  
for e in aprobados:  
    print(f"{e['nombre']}: {e['nota']}")
```

```
# Resultado:  
# Ana: 85  
# Carlos: 92  
# Elena: 78  
# Laura: 90
```

La función recorre cada estudiante, evalúa si su nota cumple el criterio mínimo, y construye una nueva lista solo con los aprobados.

# Reflexión

## ¿Qué ventajas tiene definir funciones en lugar de repetir código?

“

"El código es como un chiste: si tienes que explicarlo, es malo."

Las funciones bien nombradas se explican por sí mismas.

”

“

"La regla DRY (Don't Repeat Yourself) nos hace más eficientes."

Las funciones evitan la repetición y los errores asociados.

”

“

"Divide y vencerás: las funciones nos permiten resolver problemas complejos dividiéndolos en partes manejables."

”

# #EDCOUNIANDES

<https://educacioncontinua.uniandes.edu.co/>

*Contacto: [educacion.continua@uniandes.edu.co](mailto:educacion.continua@uniandes.edu.co)*

© - Derechos Reservados: La presente obra, y en general todos sus contenidos, se encuentran protegidos por las normas internacionales y nacionales vigentes sobre propiedad Intelectual, por lo tanto su utilización parcial o total, reproducción, comunicación pública, transformación, distribución, alquiler, préstamo público e importación, total o parcial, en todo o en parte, en formato impreso o digital y en cualquier formato conocido o por conocer, se encuentran prohibidos, y solo serán lícitos en la medida en que se cuente con la autorización previa y expresa por escrito de la Universidad de los Andes.