<codoa codo/>

FULL STACK PYTHON
Clase 26

PYTHON 2

BA

Controladores de flujo

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Clase 25

Fundamentos de Python

- Introducción a Python.
- Entorno. Hola mundo.
- Salida por pantalla: print.
- Lectura por teclado: input.
- Tipo de datos: números enteros y flotantes, texto, booleanos.
- Tipos de operadores. Aritméticos y de asignación.
- Variables.

Clase 26

Controladores de flujo

- Estructuras control.
- Condicionales: sentencia if.
- Iterativas: sentencia while y for.
- Operadores lógicos y relacionales.

Clase 27

Cadenas y Listas

- Cadenas de caracteres.
- Métodos de listas.
- f-strings
- Índices y slicing (rebanadas).
- Tipo de datos compuestos.
- Listas. Métodos.
- Tipos de datos mutables e inmutables.
- Tuplas, diccionarios, conjuntos

Estructuras de control

A diario actuamos de acuerdo a la evaluación de condiciones, incluso de manera inconsciente. Si el semáforo está en verde, cruzamos la calle. Si no, esperamos. También es frecuente evaluar más de una condición a la vez: Si llega la factura de un servicio y tengo dinero, entonces lo pago.

En Python utilizamos las **estructuras de control de flujo condicionales** para tomar decisiones similares, evaluando expresiones en las que suelen intervenir variables, para determinar qué parte del código que hemos escrito se va a ejecutar.

También disponemos de **bucles**, estructuras que permiten que un bloque de instrucciones se repita mientras que una condición sea verdadera.

Estructuras de control

En programación, las estructuras de control permiten modificar el flujo de ejecución de las instrucciones de un programa. Con ellas se puede:

- Ejecutar un grupo u otro de sentencias, según se cumpla o no una condición (**if**)
- Ejecutar un grupo de sentencias mientras se cumpla una condición (**while**)
- Repetir un grupo de sentencias un número determinado de veces (**for**)

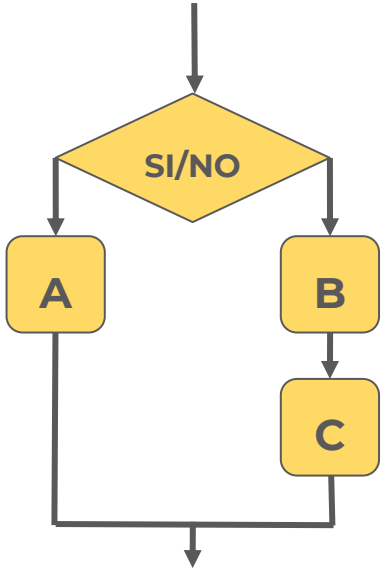
Estructuras de control

En el código de un programa podemos encontrar estructuras de los siguientes tipos:

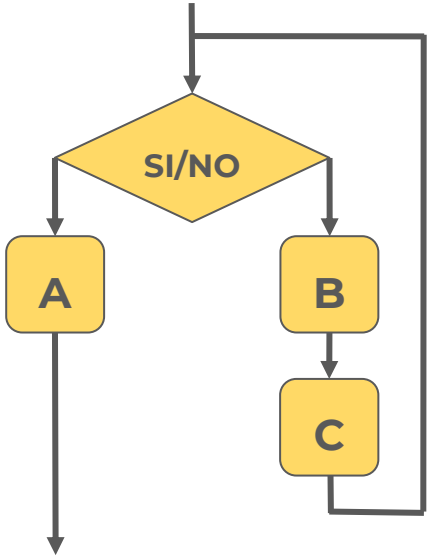
- **Secuenciales**: las instrucciones se ejecutan una después de la otra, en el orden en que están escritas, es decir, en secuencia.
- **Condicionales** (Selección o de decisión): ejecutan un bloque de instrucciones u otro, o saltan a un subprograma o subrutina según se cumpla o no una condición.
- **Iterativas** (Repetitivas): inician o repiten un bloque de instrucciones si se cumple una condición o mientras se cumple una condición.

Estructuras de control

Secuencia



Selección



Iteración

Estructuras secuenciales

Las acciones se ejecutan una seguida de la otra, es decir que se ejecuta una acción o instrucción y continúa el control a la siguiente. La ejecución es lineal y de arriba hacia abajo. Todas las instrucciones se ejecutan una sola vez y finaliza el programa.

Ejemplo de código que se ejecuta secuencialmente

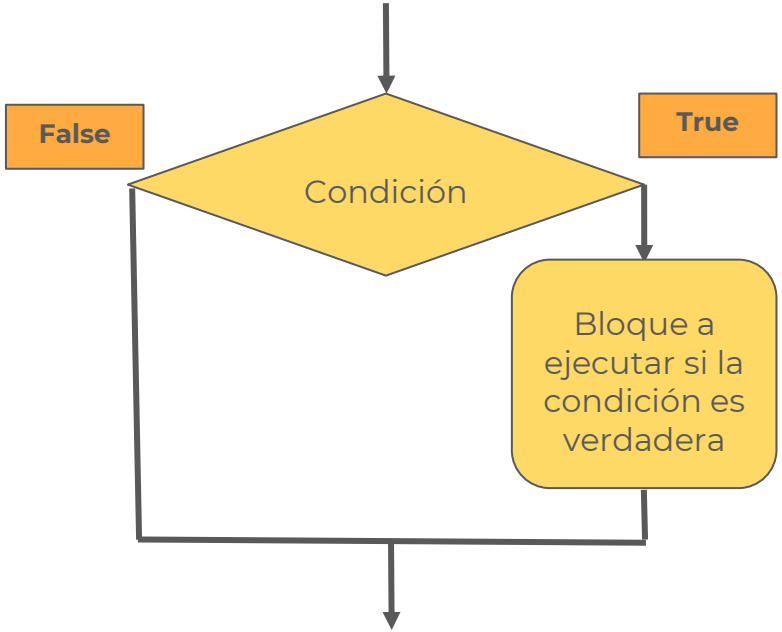
```
#Programa Suma: suma dos números enteros ingresados por teclado
nro1= int(input("Ingrese el primer número: "))
nro2= int(input("Ingrese el segundo número: "))
suma= nro1 + nro2
print("La suma es:", suma)
```

El código del ejemplo pide un número, luego pide otro, realiza la suma de ambos valores guardando el resultado en suma y finalmente muestra un mensaje por pantalla.

Estructuras condicionales

Las estructuras condicionales tienen como objetivo ejecutar un bloque de instrucciones u otro en base a una condición que puede ser verdadera o falsa. La palabra clave asociada a esta estructura es **if**.

Si la condición es **True** se ejecuta el bloque dentro del **if**. Luego, independientemente del valor de verdad de la condición, el programa continúa con la ejecución del resto del programa.



Tipos de Operadores

En la presentación anterior vimos los distintos tipos de operadores que podíamos encontrar en Python:

- Operadores de Asignación
- Operadores Aritméticos
- Operadores de pertenencia
- Operadores Relacionales
- Operadores Lógicos

A continuación, abordaremos los **operadores relacionales y lógicos**.

Operadores relacionales

Los operadores de relacionales o de comparación se utilizan, como su nombre indica, para comparar dos o más valores. El resultado de estos operadores siempre es True o False.

Donde, los operandos podrán ser variables, constantes o expresiones aritméticas.

Operador	Descripción
>	Mayor que. `True` si el operando de la izquierda es estrictamente mayor que el de la derecha; `False` en caso contrario.
>=	Mayor o igual que. `True` si el operando de la izquierda es mayor o igual que el de la derecha; `False` en caso contrario.
<	Menor que. `True` si el operando de la izquierda es estrictamente menor que el de la derecha; `False` en caso contrario.
<=	Menor o igual que. `True` si el operando de la izquierda es menor o igual que el de la derecha; `False` en caso contrario.
==	Igual. `True` si el operando de la izquierda es igual que el de la derecha; `False` en caso contrario.
!=	Distinto. `True` si los operandos son distintos; `False` en caso contrario.

Operadores lógicos

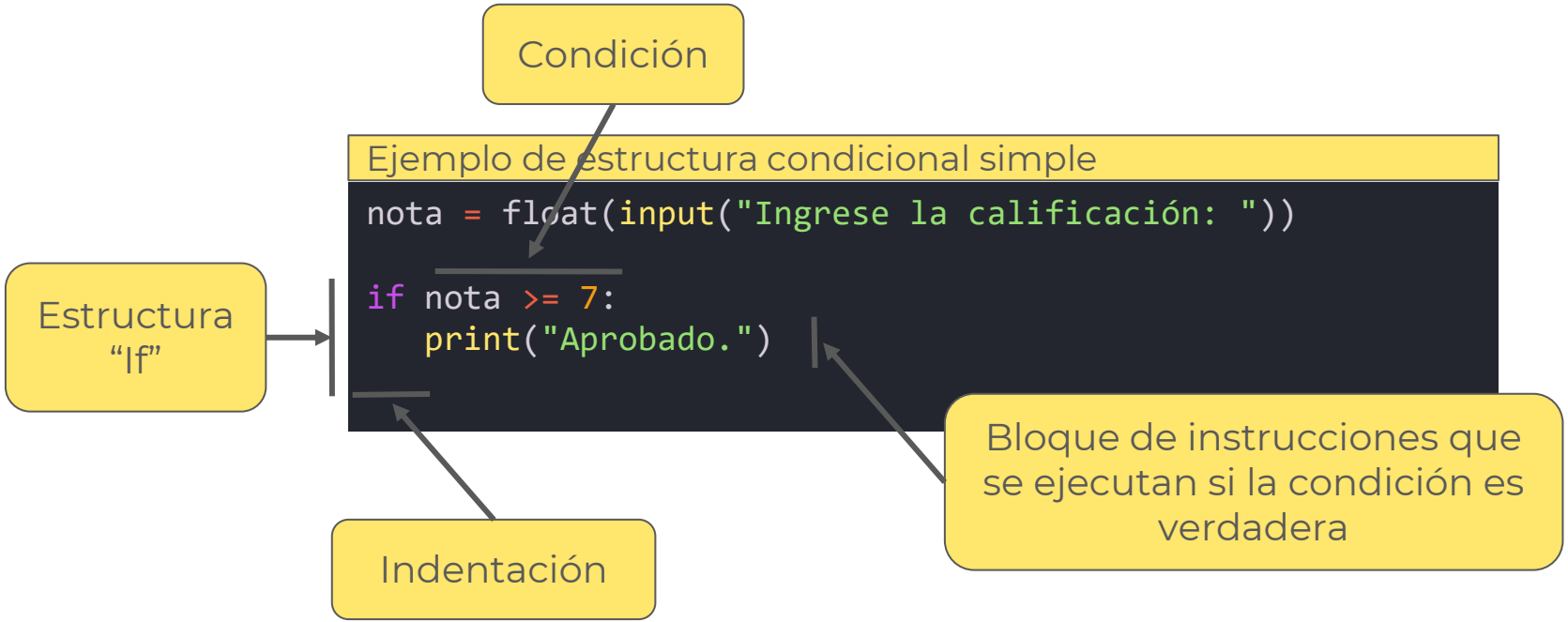
Se utiliza un **operador lógico** para tomar una decisión basada en múltiples condiciones. Los operadores lógicos utilizados en **Python** son **and**, **or** y **not**.

OPERADOR	DESCRIPCIÓN	USO
and	Devuelve True si **ambos** operandos son True	a and b
or	Devuelve True si **alguno** de los operandos es True	a or b
not	Devuelve True si el operandos False, y viceversa	not a

a y **b** son **expresiones lógicas**. Cada una de ellas puede ser **verdadera** o **falsa**.

Si **a** y/o **b** son valores numéricos, se tratan como True o False según su valor sea cero o no.

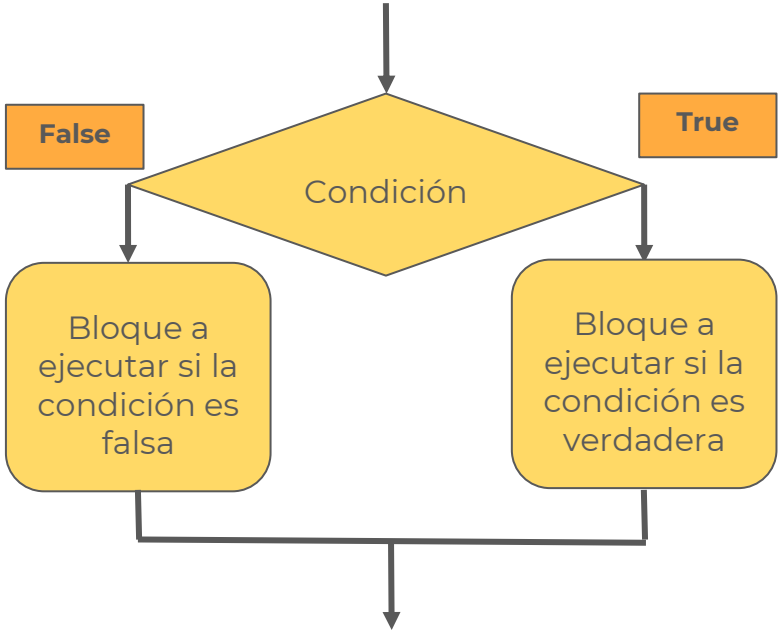
Estructuras condicionales | If



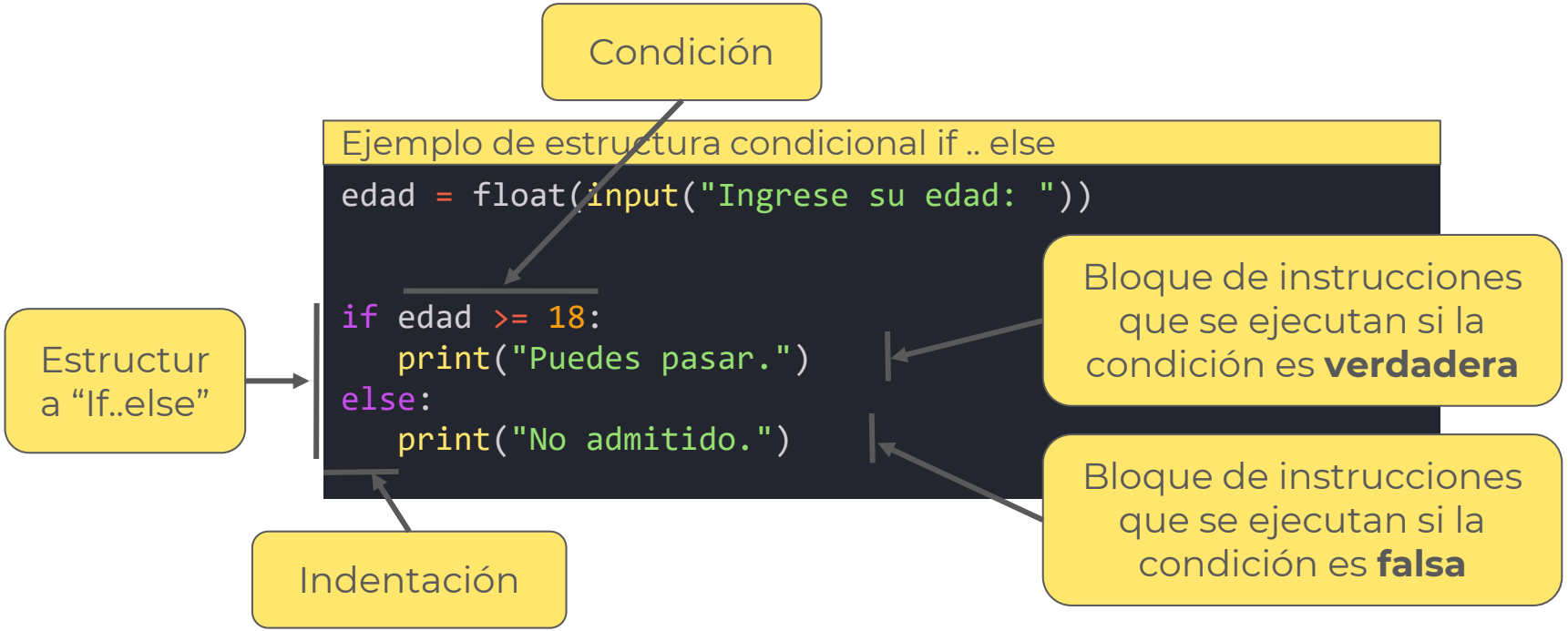
Estructuras condicionales | if .. else

En Python podemos utilizar la cláusula **else** para determinar un grupo de instrucciones que se ejecutará en caso de que la evaluación de la condición resulte ser **falsa**.

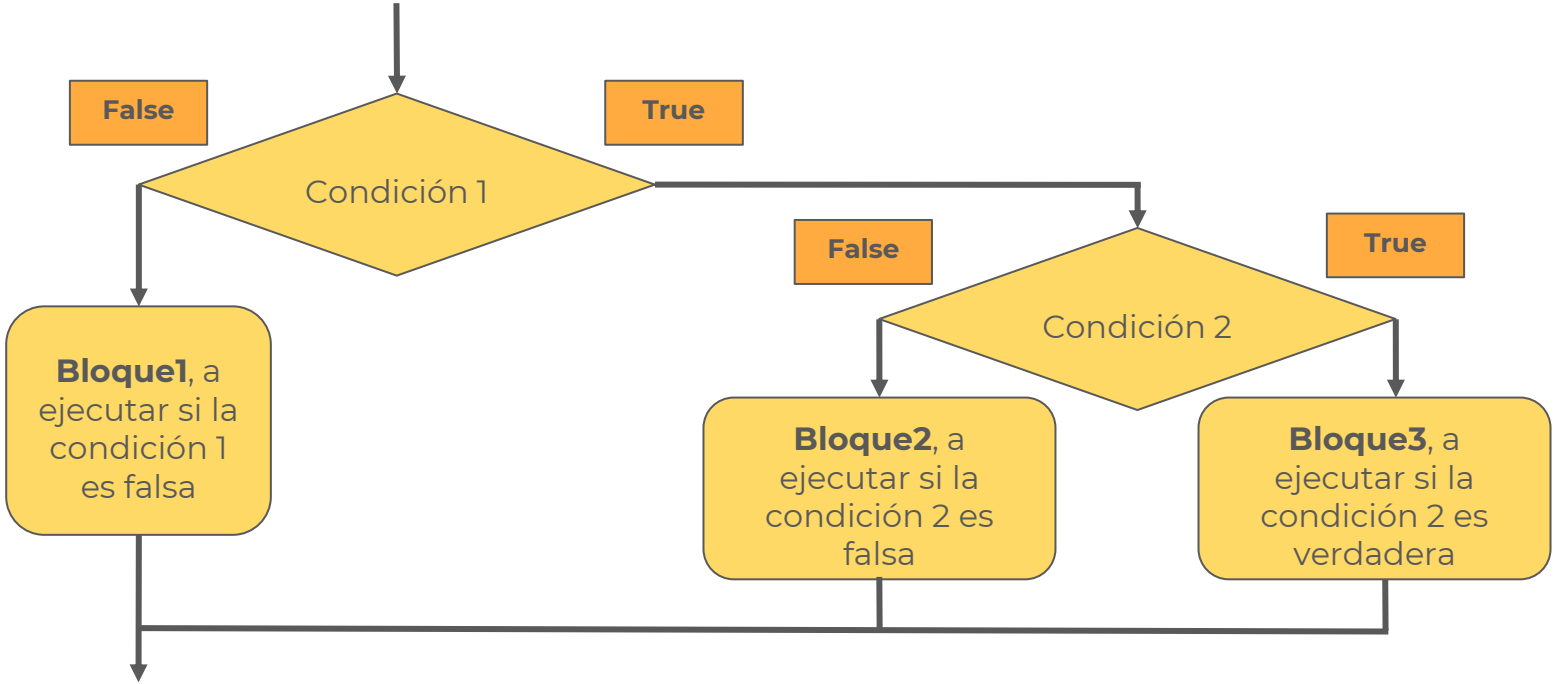
Con este agregado, una estructura **if** tiene la posibilidad de ejecutar un bloque de instrucciones u otro, dependiendo de si la condición es verdadera o falsa.



Estructuras condicionales | if .. else



Estructuras condicionales anidadas



Estructuras condicionales anidadas

En una **estructura condicional anidada**, cada **else** se corresponde con el **if** más próximo que no haya sido emparejado, y deben tener la misma indentación.

En el esquema anterior, se evalúa primero la condición 1. En caso de ser falsa se ejecuta el **Bloque3** que se encuentra en su **else**, y finaliza la ejecución de la estructura.

En caso de que la condición 1 sea verdadera, se procede a evaluar la condición 2, que se encuentra dentro (anidada) del primer **if**.

Si la segunda condición resulta verdadera se ejecuta el **Bloque3**, si resulta falsa se ejecuta el **Bloque2**.

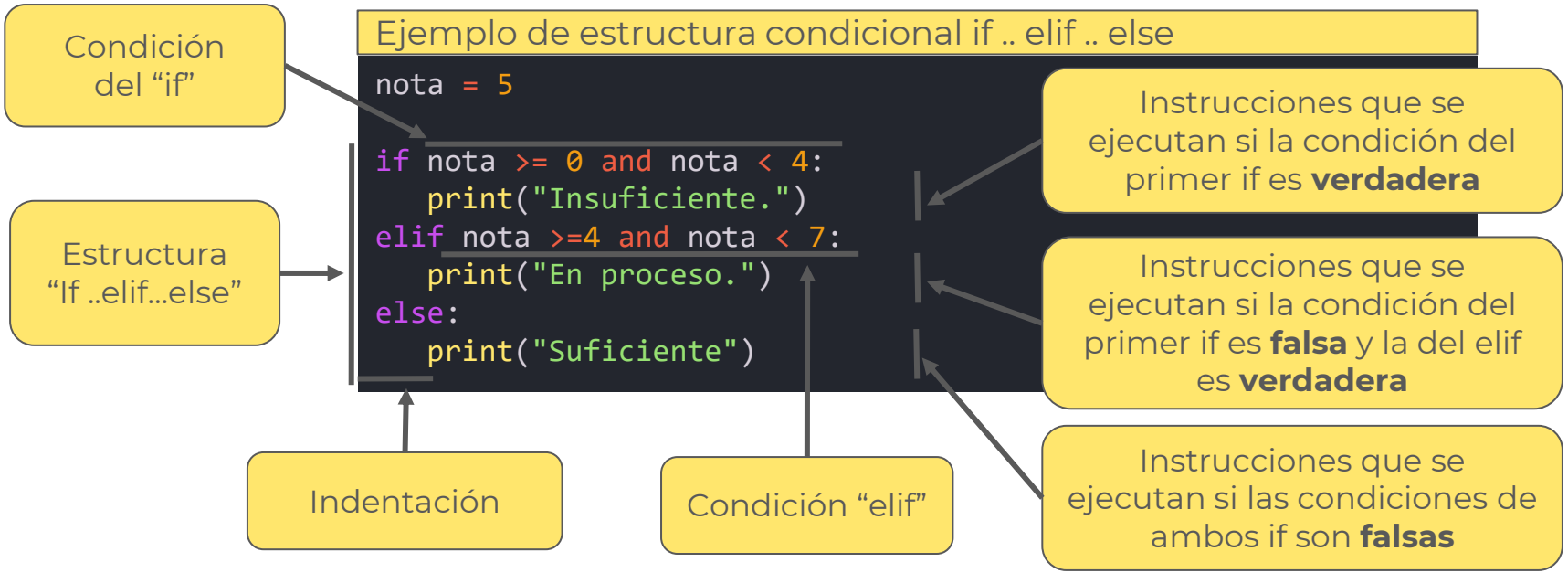
Estructuras condicionales | if .. elif .. else

A menudo solemos hacer una pregunta a partir de la respuesta de una pregunta anterior. Python tiene una estructura adecuada para implementar este comportamiento, y se conoce como **if .. elif .. else.**

La sección de código dentro del **elif** se ejecuta cuando la condición del primer **if** ha resultado ser falsa (False) y la condición del **elif** es verdadera (True). Si la condición del **elif** es falsa, entonces se ejecuta el código del bloque **else**.

Además, en situaciones más complejas se pueden utilizar **múltiples instancias de elif**, dando lugar a estructuras condicionales elaboradas y que permiten resolver prácticamente cualquier situación, aunque utilizando varios bloques de instrucciones diferentes.

Estructuras condicionales | if .. elif .. else



Estructuras condicionales

Se pueden utilizar **operadores lógicos** para tomar una decisión basada en múltiples condiciones, reduciendo la cantidad de **if**s anidados.

If con operadores lógicos

```
if (condición1) and (condición2):
if (condición1) or (condición2):
```

Las tablas de verdad muestran los valores de verdad de una proposición en función del valor lógico de sus operadores.

Cond 1	Cond 2	Y (and)
V	V	**V**
V	F	**F**
F	V	**F**
F	F	**F**

Cond 1	Cond 2	O (or)
V	V	**V**
V	F	**V**
F	V	**V**
F	F	**F**

Cond	NO (not)
V	**F**
F	**V**

Estructuras condicionales | and

Veamos un ejemplo de un **condicional** con **and**:

En un aviso del diario piden **ingenieros en sistemas con 5 años de experiencia como mínimo**, para ocupar un puesto laboral. A la convocatoria se presenta:

- Un **licenciado en sistemas** con **6 años de experiencia**: NO LO TOMAN, pues la primera condición es falsa.
- Un **ingeniero en sistemas** con **4 años de experiencia**: NO LO TOMAN, pues la segunda condición es falsa.
- Un **analista programador** con **4 años de experiencia**: NO LO TOMAN, pues las 2 condiciones son falsas.
- Un **ingeniero en sistemas** con **7 años de experiencia**: LO TOMAN, pues las 2 condiciones son verdaderas.

Estructuras condicionales | or

Veamos un ejemplo de un **condicional** con **or**:

Tengo invitados en casa y voy a comprar 1 kilo de helado. Sé que los únicos gustos que comen son chocolate o vainilla. Después de ir a varias heladerías encontré:

- Hay **chocolate** pero **no hay vainilla**. LO COMPRO, pues la primera condición es verdadera.
- Sólo hay **vainilla**, **no chocolate**. LO COMPRO, pues la segunda condición es verdadera.
- Hay **chocolate** y **vainilla**. LO COMPRO, pues las dos condiciones son verdaderas.
- Hay **crema americana** y **dulce de leche**. NO LO COMPRO, pues ninguna de las condiciones es verdadera.

Cadenas de caracteres

Python, al igual que la mayoría de los lenguajes de programación actuales, provee un tipo de datos específico para tratar las cadenas de caracteres (strings).
Se trata de un tipo de dato con longitud variable, ya que deben adecuarse a la cantidad de caracteres que albergue la cadena.
Este tipo de datos posee una buena cantidad de métodos y propiedades que facilita su uso.

Cadenas de caracteres

Una **cadena de caracteres** está compuesta por cero o más caracteres. Las cadenas pueden delimitarse con comillas simples o dobles.

Inicialización de una cadena por asignación:

```
# Definición de cadenas usando comillas dobles
dia1 = "Lunes"
x = ""          # x es un string de longitud cero

# Definición de cadenas usando comillas simples
dia2 = 'Martes'
z = "121"       # z contiene dígitos, pero es un string
```

Cadenas de caracteres

Una ventaja del hecho de poder delimitar cadenas con comillas simples o dobles es que si usamos comillas de una clase, las de otra clase puede utilizarse como parte de la cadena:

Uso de comillas simples y dobles

```
print("Mi perro 'Toby'") # Mi perro 'Toby'
print('Mi perro "Toby"') # Mi perro "Toby"
```

Una cadena puede replicarse con el operador *:

Replicación

```
risa = 'ja'
carcajada = risa*5  # jajajajaja
asteriscos = "*"*10 # **********
```

Cadenas de caracteres

También pueden usarse triples comillas simples o dobles, que proveen un método sencillo para crear cadenas usando más de una línea de código:

Cadenas delimitadas por comillas dobles o triples

```
# Definición de cadenas usando comillas dobles triples:
cadena1 = """En Python es posible definir
cadenas de caracteres utilizando más de una
línea de código"""

# Definición de cadenas usando comillas simples triples:
cadena2 = '''Por supuesto, se puede hacer
lo mismo utilizando comillas simples'''
```

Cadenas de caracteres | Concatenación

Para **concatenar** dos o más cadenas se utiliza el **operador +** (más):

Concatenación de cadenas

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

Terminal

```
Ingrese su nombre: Pedro
Hola Pedro
```

El mismo operador se usa para sumar números o concatenar cadenas. Pero no podemos utilizarlo con datos *mixtos*, porque se obtiene un error. Este error se puede evitar mediante las funciones de conversión de tipos.:

Concatenación de cadenas:

```
var1 = 3 + 5          # 8 (entero)
var2 = "3" + "5"      # 35 (cadena)
var3 = 3 + "5"        # TypeError
var4 = str(3) + "5"   # 35 (cadena)
var5 = 3 + int("5")   # 8 (entero)
```

Cadenas de caracteres | Comparación

Dos cadenas se pueden comparar mediante los operadores relacionales.

Comparación de cadenas

```
cadena1 = "Hola"
cadena2 = "Codo a Codo"
print(cadena1 > cadena2)
print(cadena1 == cadena2)
print(cadena1 < cadena2)
```

Terminal

```
True
False
False
```

La comparación es *case sensitive*, es decir, se distingue entre mayúsculas y minúsculas.

Comparación de cadenas

```
cadena1 = "Hola"
cadena2 = "hola"
print(cadena1 == cadena2)
```

Terminal

```
False
```

Cadenas de caracteres

En Python disponemos de la función **len()**, que retorna la cantidad de caracteres que contiene un string:

Función len()

```
nombre='Codo a Codo'
print(len(nombre)) #se imprime 11
```

Se accede a los elementos de la cadena utilizando subíndices:

Uso de subíndices

```
cadena = "Hola Codo a Codo"
print(cadena[0])
print(cadena[5])
print(cadena[-1])
print(cadena[-2])
```

Terminal

```
H
C
o
d
```

El primer caracter tiene subíndice cero. Si usamos subíndices negativos, se cuentan desde el final de la cadena.

Cadenas de caracteres | Métodos

Las cadenas tienen una serie de **métodos** (funciones) que simplifican el desarrollo de código. Tres de estos métodos son:

- **.upper():** devuelve la cadena con todos sus caracteres en mayúsculas.
- **.lower():** devuelve la cadena con todos sus caracteres en minúsculas.
- .**capitalize():** devuelve la cadena con su primer caracter en mayúscula y todos los demás en minúsculas.

Métodos propios de las cadenas:

```
cadena = "Codo a Codo"
print(cadena.upper()) # CODO A CODO
print(cadena.lower()) # codo a codo
print(cadena.capitalize()) # Codo a codo
```

Existen los métodos **.isupper()** e **.islower()** que devuelven True si todos los caracteres alfabéticos de una cadena están en mayúsculas o minúsculas respectivamente.

Cadenas de caracteres | Métodos

Una "**rebanada**" es un subconjunto de una cadena que se obtiene mediante la definición de un índice **inicio** y/o **fin**. El subconjunto devuelto incluye el valor del índice de inicio, pero no el valor final. Un tercer valor permite determinar un paso, que incluso puede ser negativo:

Rebanadas

```
cadena = "¡Hola mundo!"
print(cadena[6:11])   # mundo
print(cadena[2:12:2]) # oamno
print(cadena[6:])     # mundo!
print(cadena[:5])     # ¡Hola
print(cadena[:])      # ¡Hola mundo!
print(cadena[::2])    # ¡oamno
print(cadena[::-1])   # !odnum aloH¡
```

Cadenas de caracteres | in / not in

Los operadores de pertenencia se utilizan para comprobar si un caracter o cadena se encuentran dentro de otra.

OPERADOR	DESCRIPCION
in	Devuelve **True** si el valor se encuentra en una secuencia; **False** en caso contrario.
not in	Devuelve **True** si el valor no se encuentra en una secuencia; **False** en caso contrario.

Ejemplos:

```
cadena = "Codo a Codo"
print("C" in cadena)     # True
print("n" in cadena)     # False
print("Codo" in cadena)  # True
print("A" not in cadena) # True
print("o" not in cadena) # False
```

Cadenas de caracteres | for , min() y max()

Un **bucle for** puede iterar sobre una cadena, y la variable recibe en cada iteración uno de los caracteres de la misma:

For con cadenas

```
cadena = "Python"
for letra in cadena:
   print(letra)
```

Terminal

```
P
y
t
h
o
n
```

min() y **max()** devuelven el elemento con el código ASCII más pequeño o más grande respectivamente:

min() y max()

```
cadena = "Programador"
print(max(cadena))
print(min(cadena))
```

Terminal

```
r
P
```

Cadenas de caracteres | .join(), .split() y .replace()

cadena.join(separador) devuelve una cadena con el separador entre cada carácter. **cadena.split(separador)** convierte una cadena en una lista, y **cadena.replace(viejo, nuevo,max)** reemplaza una cadena por otra hasta un máximo. Si se omite max reemplaza todas las apariciones.

.join()

```
cadena = "12345"
cadena = '-'.join(cadena)
print(cadena)
#1-2-3-4-5
```

split()

```
cadena = "Codo a Codo"
lista = cadena.split(' ')
print(lista)
#['Codo', 'a', 'Codo']
```

.replace()

```
cadena = "Codo a Codo"
cadena =
cadena.replace('Codo',
'Mano')
print(cadena)
# Mano a Mano
```

Cadenas de caracteres | Detección de tipos

cadena.isalpha() devuelve True si todos los caracteres de una cadena son alfabéticos. **cadena.isdigit()** devuelve True si todos los caracteres de una cadena son dígitos. **cadena.isalnum()** devuelve True si todos los caracteres de una cadena son alfabéticos o numéricos.

.isalpha()

```
cad1 = "Python"
cad2 = "Python3"
print(cad1.isalpha())
# True
print(cad2.isalpha())
# False
```

.isdigit()

```
cad1 = "1234"
cad2 = "1234a"
print(cad1.isdigit())
# True
print(cad2.isdigit())
# False
```

.isalnum()

```
cad1 = "12Ab"
cad2 = "12Ab%"
print(cad1.isalnum())
# True
print(cad2.isalnum())
# False
```

Listas

Una **lista** es una secuencia ordenada de elementos. Pueden tener elementos del mismo tipo o combinar distintos tipos de datos, aunque esto último es poco frecuente.

Las listas en Python son un **tipo contenedor**, compuesto, y se usan para almacenar conjuntos de elementos relacionados.

Junto a las **tuplas**, **diccionarios y conjuntos**, constituyen uno de los tipos de datos más versátiles del lenguaje, con la particularidad de ser mutables. Esto último quiere decir que su contenido se puede modificar después de haber sido creadas.

Listas

Las listas se crean asignando a una variable una secuencia de elementos encerrados entre corchetes [ ] y separados por comas. Se puede crear una lista vacía, y las listas pueden ser elementos de otras listas:

Creación de listas:

```
numeros = [1,2,3,4,5] #Lista de números
dias = ["Lunes", "Martes", "Miércoles"] #Lista de strings
elementos = [] #Lista vacía
sublistas = [ [1,2,3], [4,5,6] ] # lista de listas
```

Las listas se suelen nombrar en plural. Para incluir una lista como parte de otra, basta con incluirla separada por comas de los otros elementos.

Listas | Acceso por subíndice

Las listas se pueden imprimir directamente, y el acceso a sus elementos se hace mediante subíndices. El primer elemento tiene subíndice cero. Un subíndice negativo hace que la cuenta comience desde atrás. Un subíndice fuera de rango genera un error: **out of range**

Acceso por subíndice:

```
dias = ["Lunes", "Martes", "Miércoles"]
print(dias)
print(dias[0])
print(dias[1])
print(dias[-1])
print(dias[3])
```

Terminal

```
['Lunes', 'Martes', 'Miércoles']
Lunes
Martes
Miércoles
IndexError: list index out of range
```

Listas | Recorrer listas con for y while

Es posible recorrer una lista utilizando **for** y range o **while** para generar la secuencia de índices:

For con listas

```
lista = [2,3,4,5,6]
suma = 0
for i in range(len(lista)):
   suma = suma + lista[i]
print(suma) # 20
```

For con while

```
lista = [2,3,4,5,6]
suma = 0
i = 0
while i < len(lista):
   suma = suma + lista[i]
   i = i + 1
print(suma) # 20
```

len(lista) retorna la cantidad de elementos que posee una lista y resulta muy útil, entre otras cosas, para determinar la cantidad de ciclos de un bucle.

Listas | Recorrer listas con for.. in

También se puede iterar en forma directa los elementos de la lista, sin necesidad de generar la secuencia de subíndices. En este caso la variable **i** toma el elemento de la lista:

For .. in

```
vocales = ['a','e','i','o','u']
# El bucle recorre la lista
for i in vocales:
   print(i)
```

Terminal

```
a
e
i
o
u
```

Listas | Desempaquetado y concatenado

El proceso de desempaquetado consiste en asignar cada elemento de una lista a una variable. Además, las listas pueden **concatenarse** con el operador suma:

Desempaquetado

```
dias = ["Lunes", "Martes", "Miércoles"]
d1, d2, d3 = dias
print(d1)
print(d2)
print(d3)
```

Terminal

```
Lunes
Martes
Miércoles
```

Concatenado

```
lista1 = [1,2,3]
lista2 = [4,5,6]
lista3 = lista1 + lista2
print(lista3)
```

Terminal

```
[1, 2, 3, 4, 5, 6]
```

Listas | max(), min() y sum()

- La función **max()** devuelve el mayor elemento de una lista.
- La función **min()** devuelve el menor elemento de una lista.
- La función **sum()** devuelve la suma de los elementos de una lista:

max(), min() y sum()

```
lista = [3,4,5,6]
print(max(lista))
print(min(lista))
print(sum(lista))
```

Terminal

```
6
3
18
```

Listas | in / not in y list()

Los operadores de pertenencia **in / not in** permiten determinar si un elemento está o no en una lista. La función **list()** convierte cualquier secuencia a una lista. Se puede utilizar con rangos, cadenas y otros.

in / not in

```
lista = list(range(6))
print(lista)
cadena = "Hola"
print(list(cadena))

lista2 = [3,4,5,6]
print(4 in lista2)
print(8 in lista2)
print("A" not in lista2)
```

Terminal

```
[0, 1, 2, 3, 4, 5]

['H', 'o', 'l', 'a']


True
False
True
```

Listas | .append() e .insert()

El método **append()** agrega un elemento al final de la lista

.append()

```
lista = [3,4,5]
lista.append(6)
print(lista)
```

Terminal

```
[3, 4, 5, 6]
```

insert(<pos>, <elemento>) inserta un elemento en una posición determinada:

.insert()

```
lista=[3,4,5]
lista.insert(0,2)
print(lista)
lista.insert(3,25)
print(lista)
```

Terminal

```
[2, 3, 4, 5]

[2, 3, 4, 25, 5]
```

Listas | .pop() y .remove()

pop(<posición>) Elimina un elemento en una posición determinada de la lista. Si no se pasa un argumento, pop() elimina el último elemento de la lista.
remove(<valor>) elimina un elemento en la lista, identificado por su valor.

.pop()

```
lista = [6,9,8]
lista.pop()
#Resultado: [6,9]
```

.pop(posicion)

```
lista = [3,4,5]
lista.pop(1)
#Resultado: [3,5]
```

.remove(valor)

```
lista = [3,4,5]
lista.remove(3)
#Resultado: [4,5]
```

Listas | .index(), count() y reverse()

index(<valor>) busca un valor y devuelve su posición. Admite como argumento adicional un índice inicial a partir de donde comenzar la búsqueda. **count()** devuelve la cantidad de repeticiones de un elemento, cero si no lo encuentra. Y **reverse()** Invierte el orden de los elementos de una lista.

.index(valor)

```
lista = [3,4,5]
print(lista.index(5))
#Resultado: 2
```

.count()

```
lista = [3,4,5,3,5,8,5]
print(lista.count(5))
# Resultado: 3
print(lista.count(2))
# Resultado: 0
```

.reverse()

```
lista = [3,4,5]
lista.reverse()
print(lista)
# Resulado: [5,4,3]
```

Listas | .sort() y clear()

sort() ordena los elementos de la lista, de menor a mayor. **sort(reverse=True)** ordena la lista de mayor a menor. En ambos casos, el orden depende del tipo de dato contenido en la lista. Y **clear()** elimina todos los elementos de la lista.

.sort()

```
lista = [5, 1, 7, 2]
lista.sort()
print(lista)
#Resultado: [1,2,5,7]
```

sort(reverse=True)

```
lista = [5, 1, 7, 2]
lista.sort(reverse=True)
print(lista)
#Resultado: [7,5,2,1]
```

.clear()

```
lista = [3,4,5]
lista.clear()
print(lista)
#Resultado: []
```

Estructuras repetitivas

Repiten un código dependiendo de una condición o de un contador. Si se cumple la condición se ejecuta un bloque de código y se comprueba nuevamente la condición. Pueden ser de dos clases:

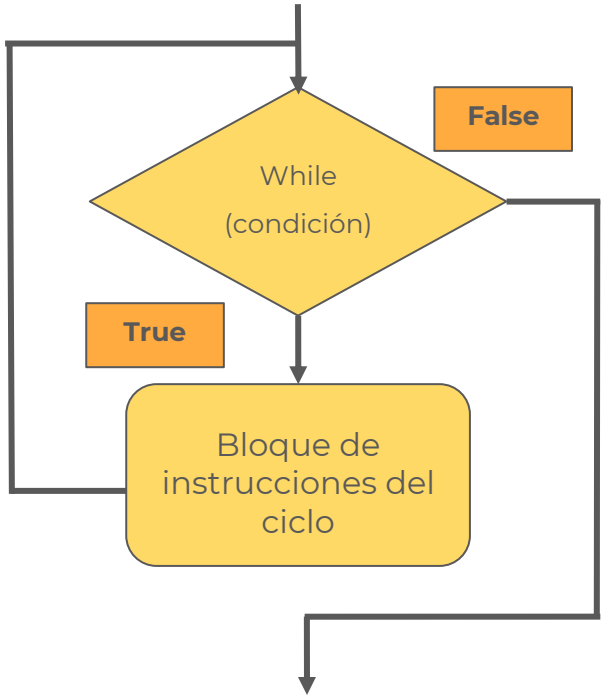
- **Ciclos Exactos:** Conocemos la cantidad exacta de repeticiones. Ese valor es aportado al iniciar el programa o por el usuario antes de que se inicie el ciclo. Los bucles **While** y **For** pertenecen a este grupo.

- **Ciclos Condicionales:** No se conoce de antemano la cantidad de repeticiones. Dependen de una condición que puede variar. Finaliza cuando la condición es falsa. Se puede repetir una vez, varias veces o ninguna vez. En este grupo se encuentra el bucle **While**.

Estructuras repetitivas | While

Ejecuta un bloque de código mientras la condición del **while** es verdadera. Finaliza cuando la condición es falsa, y no sabemos de antemano el número de veces que se va a repetir.

```
while condición:
    sentencia 1
    sentencia 2
siguiente sentencia fuera del while
```



Estructuras repetitivas | While

Podemos usar **contadores** (se incrementan o decrementan en 1 en cada ciclo) y **acumuladores** (suman algún valor en cada ciclo).

Ingresar 5 valores por teclado, obtener su suma y su promedio.

```
cont= 1
suma= 0
while cont <= 5:
    num= int(input("Ingrese un número: "))
    suma = suma + num   # Acumulamos, es equivalente suma += num
    cont = cont + 1     # Incrementamos, es equivalente cont += 1

print("La suma es:", suma)
print("El promedio es:", suma/cont)
```

Estructuras repetitivas | For

Esta estructura se utiliza cuando sabemos la cantidad de repeticiones a efectuar. Tiene el siguiente formato:

```
for i in range(inicio, fin, paso):
    sentencia1
    sentencia2
primer sentencia fuera del for
```

i: variable que incrementa su valor en **paso** unidades en cada iteración.

inicio: Es el valor inicial de i

fin: El ciclo se repite mientras i sea menor que fin.

paso: valor en que se incrementa i en cada iteración.

Si solo se escribe un número en **range**, indica el valor de **fin** (**inicio** y **paso** se asumen 1). Si se escriben dos valores, se asume que son el de **inicio** y **fin** y que **paso** es uno.

```
for i in range(fin):
for i in range(inicio, fin):
```

Se asume que inicio = 0 y paso = 1

Se asume que paso = 1

Estructuras repetitivas | For

Con el bucle **for** no necesitamos usar **contadores**, ya que la variable del ciclo asume esa función. Esto permite escribir algunos programas de una manera más compacta.

Ingresar 5 valores por teclado, obtener su suma y su promedio.

```
suma= 0
for cont in range(5):
   num= int(input("Ingrese un número: "))
   suma = suma + num   # Acumulamos, es equivalente suma += num

print("La suma es:", suma)
print("El promedio es:", suma/(cont+1))
```

Estructuras repetitivas | Break

Break permite salir de un bucle **for** o **while** en el momento que se cumpla alguna condición

break en un bucle for

```
suma= 0
for cont in range(15):
   print(cont)
   suma = suma + cont
   if cont == 3:
       break
print("La suma es:", suma)
```

Terminal

```
0
1
2
3
La suma
es: 6
```

break en un bucle while

```
cont= 0
suma= 0
while cont < 15:
   print(cont)
   suma = suma + cont
   if cont == 3:
       break
   cont = cont + 1
print("La suma es:", suma)
```

Terminal

```
0
1
2
3
La suma
es: 6
```

Sin embargo, **break** puede evitarse, y su uso no se considera una buena práctica.

Material extra

Artículos de interés

Material extra:

- Python Conditions and If statements, en w3schools
- Python While Loops, en w3schools
- Python For Loops, en w3schools

Videos:

- Estructura secuencial
- Estructura condicional
- Tablas de verdad
- If .. else e if .. else .. elif
- Uso de while
- Uso de for

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

Todo en el Aula Virtual.