1. CONTROL DE FLUJO.

¿*ESTRUCTURAS DE BLOQUE DE CÓDIGOS*?

Una estructura de bloque de código, permite agrupar instrucciones de manera controlada.

- Estructuras de control de flujo (bifurcación, bucles).
- Estructuras de cuerpo de funciones.

Una estructura de bloque, entonces, se define de la siguiente forma:

```
Orden condición :
    Código

Orden
```

En Python los bloques se delimitan por **sangrado**, utilizando siempre **cuatro espacios**. **(PEP 8: identación** Una identación de 4 (cuatro) espacios en blanco, indicará que las instrucciones identadas, forman parte de una misma estructura de control).

Cuando ponemos los **dos puntos** al final de una línea, todo lo que vaya a continuación con **un** nivel de sangrado superior se considera dentro del cuerpo. En cuanto escribimos la primera línea con un nivel de sangrado inferior, hemos cerrado el condicional.

Si no seguimos esto a rajatabla Python nos dará errores; es una forma de forzar a que el código sea legible.

¿*SENTENCIAS NECESARIAS EN UN BLOQUE DE CÓDIGO*?

`pass` ---------- La sentencia pass / *pass*. Pasar una declaración nula, una declaración que no hará nada.

Las estructuras de control de flujo, son aquellas que nos permiten evaluar si una o más condiciones se cumplen, para decir qué acción vamos a ejecutar.

La evaluación de condiciones, solo **puede arrojar** 1 de 2 resultados: **verdadero o falso** (`True` o `False`).

Para describir la evaluación a realizar sobre una condición, se utilizan **operadores relacionales** (o de comparación), o lógicos.

2. ESTRUCTURAS DE CONTROL DE FLUJO (BIFURCACIÓN/ALTERNATIVAS).

Las Estructuras de control alternativas, permiten direccionar el flujo del programa, dependiendo del resultado de una condición.

```
if condición1 :                 #Si
    codigo asociado true1
[elif condición2 :              #Sino, entonces si ..
    codigo asociado condición2 ] . . .
[else :                         #Sino
    codigo asociado false1 ]
```

`if` ---------- La orden if / *if*.Da paso a la sentencia de bifurcación.
+ Solo puede haber una.

`elif` ---------- La orden elif /*elif*. Opcional. Se usa en sentencias de bifurcación.
La palabra reservada 'elif' es una abreviación de 'else if', y es útil para

evitar un sangrado excesivo.
+ Puede haber de cero o N bloques elif,

else ---------- La orden else /*else*. Opcional. Utilizado en sentencias de bifurcación (también en bucles).
+ Puede haber cero o uno else.

¿*ALGUNOS FORMATOS*?

Además del formato en bloque, encontramos formatos en una misma línea, estos son:

- Si solo tiene una instrucción que ejecutar, puede colocarla en la misma línea que la instrucción if.

```
if condición : codigo true
```

- Si solo tiene que ejecutar una instrucción, una para sí y otra para otra cosa, puede ponerlo todo en la misma línea.

```
código true if condición else código false
```

- Una línea en caso de otra cosa, con 3 condiciones:

```
código true1 if condición1 else código true2 if condición2 else código false2
```

¿*ALTERNATIVA MÚLTIPLE- SWITCH O CASE DE OTROS LENGUAJES*?

La secuencia if ... elif ... elif ... sustituye las sentencias de bifurcación multiple (switch).

¿*IF ANIDADOS*?

En Python también se pueden anidar los if.

3. ESTRUCTURAS DE CONTROL DE FLUJO (BUCLES/CICLICAS/REPETITIVAS/ ITERATIVAS).

Las estructuras de control iterativas (también llamadas cíclicas o bucles), nos permiten ejecutar un mismo código, de manera repetida, mientras se cumpla una condición.

En Python se dispone de dos estructuras cíclicas:

- El bucle `while`
- El bucle `for`

En Python también se pueden anidar los bucles.

El "bucle interno" se ejecutará una vez para cada iteración del "bucle externo":

3.1. WHILE.

El bucle while, permite repetir un bloque de código “mientras que se cumpla una condición”.

while ---------- La orden while / *while*.. Da paso a un bucle que se repite "mientras que" se cumpla una condición.
+ Admite las sentencias: **break** y **continue**.
+ Pueden darse **while** anidados.

```
Entrada-condición
while condición :        # mientras se cumpla
    código asociado...
    [break]
    [continue]
    Cambio-condición
[else:
    código asociado...]
```

+ Se puede interrumpir /salir del bucle con la sentencia **break**.
+ Con **continue** detenemos la iteración actual (saltar las siguientes sentencias del bucle) y continuamos con la siguiente.
+ El código del bloque **else** se ejecutará una vez finalizado el bucle, siempre que no haya sido interrumpido con **break**.

3.2. FOR IN

El bucle **for**, nos permitirá iterar/recorrer datos de cualquier secuencia (cadena, tuplas, listas, set, diccionarios, etc., en el orden que aparecen en la secuencia, ejecutando para cada iteración un grupo de sentencias.

Nota: Si necesitas modificar la secuencia sobre la que estás iterando mientras estás adentro del ciclo (por ejemplo para borrar algunos ítems), se recomienda que hagas primero una copia. Iterar sobre una secuencia no hace implícitamente una copia.

for----------La orden for / *for*. Da paso a un bucle que se repite un determinado número de veces, según formato.
+ Admite las sentencias: **break** y **continue**.
+ Pueden darse **for** anidados.

```
for variable in [colección | funcion(x)] :
    código asociado...
    [break]
    [continue]
    [código asociado]...
 [else:
    código asociado...  ]
```

+ Se puede interrumpir el bucle a la mitad con la sentencia **break**.
+ Con **continue** detenemos la iteración actual (saltar las siguientes sentencias del bucle) y continuamos con la siguiente.
+ El código del bloque **else** se ejecutará una vez finalizado el bucle, siempre que no haya sido interrumpido con **break**.

¿*ALGUNOS FORMATOS*?

A) in colección

```
for variable in coleccion
```

● Se repite tantas veces como número de elementos de la colección (secuencia/dato "iterable"-cadenas, tuplas, listas, set, diccionarios, etc.-).

+ La variable va tomando los valores de los elementos de la colección

B) in range()

```
for variable in range(x)
```

● Se repite tantas veces como la secuencia de números generados por la función.

Utilizar cuando queremos "emular" el funcionamiento del f**or** en otros lenguajes.

+La variable va tomando los valores de cada uno de los números de la secuencia generada.

range(*x*) ----------La función incorporada range(). Devuelve una secuencia de números, comenzando desde 0 e incrementando en 1 (por defecto).

```
[variable=]range([inicio,]final[,incremento])
```

+ **final**: Un número entero, indica el número de ítems que contendrá la secuencia.
  -Empezara siempre por 0, se incrementa en 1, y finaliza en `numero -1`.

+ `inicio`: Un número entero, indica en qué valor comenzar.
  - Por omisión, el valor predeterminado es 0.
+ `incremento`: Un número entero, indica el incremento.
  - Por omisión, el valor predeterminado es 1.

  - Puede ser negativo.

C) in enumerate()

```
for variable in enumerate(x)
```

● Se repite tantas veces como número de elementos de la colección (secuencia/dato "iterable"-cadenas, tuplas, listas, set, diccionarios, etc.-).

+ La variable va tomando los valores de los elementos de la colección

enumerate(*x*) ----------La función incorporada enumerate(). Devuelve un objeto <**class 'enumerate'**> .

+ A partir de la colección/secuencia, agrega un contador, de tal forma que devuelve un tipo <class 'tuple'> con dos elementos, donde el elemento 0 es el contador (la enumeración), y el elemento1 es el contenido del elemento de la colección correspondiente.

```
[variable =]enumerate(coleccion[,inicio])
```

+ **coleccion**: cualquier objeto iterable.
+ `inicio`: Un número entero, indica en qué valor inicial del "contador" del objeto <**class 'enumerate'**>.
  - Por omisión, el valor predeterminado es 0.
  -También puede ser negativos

D) in zip()

```
for variable in zip((x))
```

● Se repite tantas veces como número de elementos de la colección con menos elementos.

+ La variable va tomando los valores de los elementos de las colecciones

zip(*x*) ----------La función incorporada zip(). Devuelve un objeto **<class 'zip'>**.

+ A partir de las colección, enviadas como parámetros, devuelve un tipo <class 'tuple'> con tantos elementos como colecciones.
  Si las colecciones tienen diferentes longitudes, se tomara como longitud la de la colección menor.

```
[variable=]zip(coleccion1[,coleccion2]...)
```

+ `coleccion1-N`: Objetos iterables que se unirán.

3.3. ORDENES DE RUPTURA (SALIDA/CONTINUAR).

En las estructuras de flujo anteriores, se permite utilizar ordenes de ruptura del flujo como son:

`break` ---------- La orden break / *break*, termina el bucle **`for`** o **`while`** más anidado.

`continue` ---------- La orden continue/, *continue*, termina la iteración actual y continua con la siguiente iteración del bucle **`for`** o **`while`** más anidado.