¿Qué es un condicional?

En los lenguajes de programación como Python, los condicionales son instrucciones que le damos a un programa para que ejecute un bloque de código u otro, dependiendo de unos determinados requisitos. Las sentencias condicionales que permiten que un programa ejecute un código si se cumplen las condiciones requeridas son: ***If, Elif y Else***.
Veamos un ejemplo de un condicional simple. Primero creamos una variable de edad:

```
edad = x (cifra variable)
```

Ahora le decimos al programa, si la edad es menor de dieciocho años, imprime esta frase:

```
edad = 15
if edad < 18:
   print(“La edad está por debajo del límite”)
```

Si queremos introducir una segunda instrucción, por ejemplo, en el caso de que la edad fuera la adecuada, utilizamos la sentencia ***else*** junto con la sentencia ***if***:

```
edad = 35
if edad < 18:
   print(“La edad está por debajo del límite”)
else:
  print(“La edad es adecuada”) #imprime: La edad es adecuada
```

En el primer recuadro la edad del usuario es menor de dieciocho, por lo tanto el requisito ***if*** es ***True*** *(verdadero)* y el programa ejecutara la instrucción dada por ***if***. En el segundo recuadro la edad es mayor de dieciocho lo cual hace que el requisito ***if*** sea ***False*** *(falso)*, por lo que el programa pasara a evaluar el siguiente requisito y ejecutara la instrucción de ***else***. Las sentencias ***if*** e ***if-else*** son solo útiles para situaciones binarias. En el caso de un problema condicional múltiple se utiliza la sentencia ***if-elif-else***. En primer lugar se comprueba la condición de la sentencia ***if***. Si es falsa se evalúa la sentencia ***elif***, y si esta también es falsa entonces se evalúa y ejecuta la sentencia ***else***. Veamos un ejemplo:

```
edad = 35
if edad < 18:
   print(“La edad está por debajo del límite”)
elif edad > 40:
   print(“La edad está por encima del límite”)
else:
  print(“La edad es adecuada”) #imprime: La edad es adecuada
```

En el siguiente recuadro podemos ver también un ejemplo de utilización de sentencias condicionales con cadenas de caracteres (*Strings*):

```
nombre = “Ana Martinez”
if nombre == “Ana Gonzalez”:
    print(“Autorizado”)
else:
    print(“No autorizado”) #imprime: No autorizado
```

¿Cuáles son los diferentes tipos de bucles en Python? ¿Por qué son útiles?

Al igual que los condicionales, los bucles (*Loops*) son comunes en los lenguajes de programación. Un bucle es una herramienta que nos permite ***iterar*** una porción de código tantas veces como queramos. El término *iterar* significa “realizar una acción varias veces”, y cada repetición de esa acción se conoce como ***iteración***. Un bucle hace que un programa implemente una *iteración* mientras la condición declarada se cumpla, es decir, los bucles dependen de que una condición sea ***True*** *(verdadera)* o ***False*** *(falsa)* para ser ejecutados en cierta manera. El programa sale del bucle cuando termina de iterar o cuando la condición llega a ser falsa.

Python incluye únicamente dos tipos de bucles: ***For*** y ***While***. La principal diferencia entre el bucle ***For*** y el bucle ***While*** es que ***For*** tiene un numero de iteraciones definido; mientras que ***While*** tiene un número de iteraciones indefinido; es decir, puede ejecutar el código indefinidamente, resultando en un **bucle infinito**, a no ser que le indiquemos cómo y cuándo detener el proceso. Un bucle infinito puede dar lugar a errores (*bugs*) en el programa. Veamos un ejemplo de cada bucle:

FOR

En Python, hay principalmente dos tipos de bucles ***For:*** el bucle *For* tradicional y el bucle *For* utilizando la función *range( )*. El bucle *For* tradicional itera sobre los elementos de una secuencia (como una lista, tupla, diccionario, etc.) uno por uno y ejecuta la orden que le demos a continuación. Veamos un ejemplo:

```
numeros = [32, 14, 7, 90, 1]
for num in numeros:
     print (num)  # imprime: 32 14 7 90 1
```

La función *range( )* sirve para crear un rango de números en un bucle ***For***. Tiene tres parámetros con valores por omisión, por lo que puede tener uno, dos o tres argumentos: start _ El primer numero del rango, stop _ El último numero del rango y step _ El valor (pasos) que se suma al primer numero para conseguir los números consecutivos. Veamos unos ejemplos:

```
# con dos argumentos, comenzando en 1, parar en 11
for num in range(1, 11):
        print(num) #imprime: 1 2 3 4 5 6 7 8 9 10

# con tres argumentos, comenzando en 1, parar en 22, saltando de 3 en 3
for num in range(1, 22, 3):
        print(num) #imprime: 1 4 7 10 13 16 19
```

** Nótese que en las listas y tuplas, los elementos se comienzan a contar desde cero, por lo que si queremos obtener una lista de números del 1 al 10, éste último incluido, el rango a introducir deberá ser de 1 a 11.*

WHILE

El bucle ***While*** en Python se utiliza para ejecutar un bloque de código mientras una condición sea verdadera. Veamos unos ejemplos:

El primer bucle imprimirá los números del 0 al 6. La condición *contador < 7* se evalúa antes de cada iteración. El segundo bucle imprimirá la lista de nombres de perros. La sintaxis de la última línea del código: *contador += 1* es equivalente a escribir: *contador = contador + 1*.

```
# Ejemplo 1
contador = 0
while contador < 7:
    print(contador)
    contador += 1 #imprime: 0 1 2 3 4 5 6

# Ejemplo 2
mis_perros = ['Luna', 'Princesa', 'Oscar', 'Dama']
contador = 0
while contador < len(mis_perros):
  print(mis_perros[contador])
  contador += 1 #imprime: Luna Princesa Oscar Dama
```

En cada iteración del bucle, aumentamos el valor de *contador* en 1. Esto es importante para evitar un bucle infinito, ya que eventualmente *contador* alcanzará un valor que hará que la condición *contador < 7* sea falsa y el bucle se detendrá.

El ciclo ***For*** es una herramienta muy útil en Python que permite iterar sobre objetos iterables y realizar operaciones repetitivas de manera sencilla y eficiente. El bucle ***While*** puede realizar la misma tarea varias veces mientras una condición determinada sea verdadera.

¿Qué es una lista por comprensión en Python?

Las **listas de comprensión** en Python son una forma concisa y elegante de crear listas. Permiten generar listas utilizando una sintaxis más compacta que los bucles tradicionales. Básicamente es como sintetizar un conjunto de bucles *For* y condicionales en una sola línea de código. Por ejemplo, queremos crear una nueva lista a partir de una lista ya existente (numeros = [1, 2, 3, 4, 5, 6]). Los elementos de la nueva lista serán los valores numéricos de la lista uno, +1 (resultado = [2, 3, 4, 5, 6, 7]).

En el siguiente recuadro veremos primero la forma tradicional de escribir el código seguido de un ejemplo de una lista de comprensión:

```
# crear una lista a partir de una lista existente donde sumemos +1 a cada
elemento de la lista ya dada

# modo tradicional

numeros = [1, 2, 3, 4, 5, 6]
resultado = []

for num in numeros:
    resultado.append (num + 1)

print(resultado) #imprime: 2 3 4 5 6 7

# lista de comprensión

numeros = [1, 2, 3, 4, 5, 6]
resultado = [num + 1 for num in numeros]
print(resultado) #imprime: 2 3 4 5 6 7
```

¿Qué es un argumento en Python?

En Python, **los argumentos** son los valores que se pasan a una función cuando se llama. Si entendemos que los parámetros son las variables dentro de los paréntesis de una función, los argumentos son los que proporcionan unos valores para esos parámetros. Pueden ser *posicionales*, de *palabra clave* o *argumentos por defecto*. Veamos algunos ejemplos de estos argumentos en Python:

```
# Argumentos posicionales

def saludar (nombre, edad):
     print(f"Hola {nombre}, tienes {edad} años.")
saludar("Juan", 30) #imprime: Hola Juan, tienes 30 años
```

Cuando invocamos la función, en los *argumentos posicionales* se sigue el orden dado por

los parámetros, es decir, el primer argumento se asignara al primer parámetro entre paréntesis, el segundo al segundo, el tercero al tercero, etc.

```
# Argumentos de palabra clave

def saludar (nombre, edad):
     print(f"Hola {nombre}, tienes {edad} años.")
saludar(nombre="María", edad=25) #imprime: Hola María, tienes 25 años
```

Los *argumentos de palabra clave*, o argumentos con nombre, son valores que cuando se pasan a una función son identificables por nombres de parámetros específicos. En este caso dejamos claro al invocar la función, que el valor "*Maria"* por ejemplo, ha de asignarse al parámetro "*nombre",* y el valor *"25"* corresponde al parámetro *"edad"*. De manera que el orden de invocación es irrelevante.

```
# Argumentos por defecto

def saludar (nombre="Usuario", edad=18):
     print(f"Hola {nombre}, tienes {edad} años.")
saludar() #imprime: Hola Usuario, tienes 18 años
```

Los *argumentos por defecto* son valores que se asignan directamente a los parámetros de una función en caso de que al invocar esa función no se proporcionen valores para ellos.

¿Qué es una función Lambda en Python?

Una función **lambda** es una función anónima (definida sin nombre) que puede tomar cualquier número de argumentos pero, a diferencia de las funciones normales, evalúa y devuelve solo una expresión. Son útiles cuando necesitas una función rápida para una operación simple. Las funciones lambda se declaran dentro de una variable, y la sintaxis es la siguiente: *variable = lambda <argumentos> : expresión.* Veamos algunos ejemplos sencillos:

En el primer ejemplo los parámetros serian ***x, y*** y los valores otorgados a esos parámetros ***3, 5.*** La expresión seria la suma de ambos parámetros. En el segundo ejemplo vemos solo hay un parámetro, ***x***, el valor dado a éste es ***4*** y la expresión la operación de elevar ese numero al cuadrado.

```python
# Sumar dos números

suma = lambda x, y: x + y
print(suma(3, 5))  #imprime: 8

# Elevar un número al cuadrado

cuadrado = lambda x: x ** 2
print(cuadrado(4))  #imprime: 16
```

¿Qué es un paquete pip?

Los paquetes PIP en Python son colecciones de código reutilizable que se pueden instalar y gestionar fácilmente utilizando la herramienta PIP. Estos paquetes pueden contener módulos, funciones y otros recursos que pueden ser utilizados en tus proyectos de Python para agregar funcionalidad adicional. Algunos ejemplos populares de paquetes PIP en Python incluyen:

1. NumPy _ Para computación numérica y operaciones matemáticas.
2. Pandas _ Para análisis y manipulación de datos.
3. Matplotlib _ Para visualización de datos.
4. TensorFlow o PyTorch_Para machine learning y deep learning.
5. Django o Flask _ Para desarrollo web.
6. Requests _ Para realizar peticiones HTTP.
7. Beautiful Soup _ Para hacer scraping web.
8. SciPy _ Para herramientas y algoritmos científicos.
9. Scikit-learn _ Para machine learning y análisis de datos.
10. NLTK (Natural Language Toolkit) _ Para procesamiento de lenguaje natural.