¿Qué es un condicional?

Un condicional en programación es una estructura de control que nos permite ejecutar un bloque de código si se cumple una condición determinada. En Python, los condicionales se implementan mediante palabra reservada '**if**', seguida de una condición, tiene el siguiente aspecto:

```
if condición:
    código_a_ejecutar_si_es_verdadero
```

La sentencia inicial (la que corresponde al '**if**') debe ir seguida de dos puntos, y todas las líneas que dependen de ella deben estar indentadas un nivel (preferiblemente cuatro espacios). Cuando la expresión que sigue a la orden **'if'** se evalúa y retorna un valor '**True**', se ejecuta el bloque de código. Si la expresión devuelve **'False'**, el bloque no se ejecuta y el programa continúa con la instrucción siguiente.

Ejemplo

```
edad = 18
if edad >= 18:
    Print ("Eres mayor de edad")
Salida #"Eres mayor de edad"
```

Cuando el contenido del bloque solo tiene una línea, se puede poner a continuación del '**if**', en lugar de debajo:

Ejemplo

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

Se puede usar la palabra reservada **'else'** para indicar las expresiones que se ejecutarán si la condición del **'if'** no se cumple. Tiene el siguiente aspecto:

```
if condición:
  código_a_ejecutar_si_es_verdadero
else:
  código_a_ejecutar_si_es_falso
```

Ejemplo

```
edad = 16
if edad >= 18:
    print ("Eres mayor de edad")
else:
    print ("No eres mayor de edad")
```

En este ejemplo, se imprimirá "No eres mayo de edad" porque la condición del **'if'** no se cumple.

También hay una forma de hacer construcciones condicionales más elaboradas gracias a la orden **'elif'** (contracción de else+if), esta permite añadir un bloque que se ejecutará si se cumple la condición específica y no se cumplen las anteriores.

```
if condición1:
    código_a_ejecutar_si_condición_1_es_verdadera
elif condición_2:
    código_a_ejecutar_si_condición_2_es_verdadera
else:
    código_a_ejecutar_si_ninguna_condición_es_verdadera
```

Ejemplo

```
Nota = 8,5
if nota >= 9:
    print ("Tienes una A")
elif nota >= 8:
    print ("Tienes una B")
elif nota >= 7:
    print ("Tienes una C")
else:
    print ("Hay que mejorar")
```

En el ejemplo, si la variable nota es 8,5, se imprimirá "Tienes una B" porque cumple la condición 'nota >= 8'

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

Los bucles permiten repetir un bloque de código un número determinado de veces o hasta que se cumpla una condición. Son fundamentales en la programación porque nos permiten automatizar y ejecutar tareas repetitivas de manera eficiente. Los tipos de bucles más comunes en Python son:

1. **Bucle 'for'**

   El bucle **'for'** itera sobre una secuencia de elementos (como una lista, una tupla, un diccionario, un conjunto o un rango de números). Cada elemento de la secuencia se asigna a una variable temporal y se ejecuta el bloque de código correspondiente para cada elemento.

   Ejemplo

```
Nombres = ["Juan", "María", "Pedro"]
for nombre in nombres:
    print (nombre)
# Salida:
Juan
Maria
Pedro
```

   En el ejemplo, el bucle **'for'** itera sobre la lista **'nombres'** e imprime cada nombre en la lista.

   Ejemplo

```
for i in range (1,6):
    print (i)
#Salida:
1
2
3
4
5
```

   Este bucle imprimirá los números del 1 al 5. La función **'range (1,6)'** genera una secuencia de números desde el 1 hasta el 5 (el 6 no se incluye).

2. **Bucle 'while'**

   El bucle **'while'** se ejecuta mientras se cumpla una condición. La condición se evalúa antes de cada iteración y, si es **'True'**, el bloque de código se ejecuta. Si es **'False'**, el bucle termina.

Ejemplo

```
contador = 1
while contador <= 10:
    print (contador)
    contador += 1
#Salida:
1
2
.
10
```

En este ejemplo, el bucle **'while'** imprimirá los números del 1 al 10. La variable **'contador'** se incrementa en 1 en cada iteración hasta que alcanza el valor de 11, momento en el que la condición **'contador <=10'** ya no se cumple y el bucle termina.

¿Por qué son útiles los bucles?

Se pueden mencionar varias razones:

- **Automatización de tareas repetitivas**: permiten ejecutar un bloque de código múltiples veces sin necesidad de escribir el código repetidamente.

Ejemplo

```
for i in range(100):
    print ("Este mensaje se imprimirá 100 veces")
```

- **Procesamiento de datos en colecciones:** facilita la iteración y manipulación de elementos en listas, tuplas, diccionarios, y conjuntos.

Ejemplo

```
precios = [100, 200, 300, 400]
total = 0
for precio in precios:
    total += precio
print ("El total es: ", total)
#Salida: El total es: 1000
```

- **Condiciones dinámicas:** los bucles **'while'** permiten ejecutar código hasta que cumpla una condición específica, lo que es útil para situaciones en las que no se sabe de antemano cuántas iteraciones serán necesarias.

Ejemplo

```
respuesta = ""
while respuesta != "si":
      respuesta = input ("¿Has terminado tu tarea? (si/no): ")
print ("¡Estupendo!")
#Salida:
¿Has terminado tu tarea? (si/no): no"
¿Has terminado tu tarea? (si/no): si"
¡Estupendo!
```

- **Eficiencia en la programación:** reducen la cantidad de código y hacen que los programas sean más fáciles de leer, mantener y escalar.

Ejemplo

```
números= [1, 2, 3, 4, 5]
cuadrados = [número ** 2 for número in números]
print(cuadrados)
# Salida: [1, 4, 9, 16, 25]
```

En resumen, los bucles **'for'** y **'while'** son herramientas poderosas y esenciales en Python que permiten manejar de manera eficiente las tareas repetitivas y el procesamiento de colecciones de datos, haciendo que la programación sea más efectiva y flexible.

¿Qué es una lista por comprensión en Python?

Una lista por comprensión es una manera concisa y legible de crear listas en Python. Utiliza una única línea de código para generar listas a partir de iterables, aplicando una operación a cada elemento de estos. Este enfoque no solo hace que el código sea más compacto, sino también más fácil de leer y mantener.

Imaginemos que tenemos una lista con los días de la semana en minúsculas, y queremos crear, a partir de ella, otra que tenga la primera letra de cada día en mayúsculas. Sin usar listas de comprensión, podríamos hacerlo de la siguiente manera:

Ejemplo

```
dias_semana = [“lunes”, “martes”, “miércoles”, “jueves”, “viernes”, “sábado”, “domingo”]
semana_mayus =[]
for día in dias_semana:
      semana_mayus.append(día.capitalize())
print(semana_mayus)
#salida: ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado', 'Domingo']
```

En este código se ha creado la lista **‘semana_mayus’** vacía y luego, usando un bucle **‘for’**, se ha recorrido la lista original para aplicar una transformación a cada elemento y añadir el resultado a la lista vacía por medio del método **‘append’.**

La misma operación puede realizarse de forma más concisa utilizando **una lista por comprensión:**

Ejemplo

```
dias_semana= [“lunes”, “martes”, “miércoles”, “jueves”, “viernes”, “sábado”, “domingo”]
semana_mayus =[día.capitalize()for día in dias_semana]
print(semana_mayus)
#salida: ['Lunes', 'Martes', 'Miercoles', 'Jueves', 'Viernes', 'Sabado', 'Domingo']
```

Supongamos que tenemos una lista de números y queremos crear a partir de ella otra que contenga los mismos elementos, pero multiplicados por dos:

Ejemplo

```
numeros = [1,2,3,4,5]
numeros_doblados = []
for numero in numeros:
    numeros_doblados.append(numero * 2)

print(numeros_doblados)
#salida: [2, 4, 6, 8, 10]
```

Con listas por comprensión, el mismo resultado se puede obtener de la siguiente manera:

Ejemplo

```
numeros = [1,2,3,4,5]
numeros_doblados = [numero*2 for numero in numeros]

print(numeros_doblados)
#salida: [2, 4, 6, 8, 10]
```

Sintaxis de las Listas por Comprensión

Lista = [*OPERACIÓN* **for** VARIABLE **in** ITERABLE **]**

La lista por comprensión se declara entre corchetes, []. Dentro de esos corchetes se define el bucle **'for'** del modo habitual, pero sin terminar en dos puntos, precedido de la operación que queremos aplicar a cada uno de los valores que retorne el bucle, y cuyo resultado será uno de los elementos de la lista final.

Se puede usar cualquier iterable para crear una lista por comprensión. Por ejemplo, se puede definir una lista a partir de un diccionario.

Ejemplo

```
diccionario = {'a': 1, 'b': 2, 'c': 3}
nueva_lista = [clave + str(valor) for clave, valor in diccionario.items()]

print(nueva_lista)
#salida: ['a1', 'b2', 'c3']
```

Igualmente, se pueden agregar condiciones a las listas por comprensión para **filtrar los elementos que quieres incluir**. Esto se hace agregando una cláusula **'if'** al final de la expresión.

Ejemplo

```
numeros = ['1,2,3,4,5,6,7,8,9,10]
pares = [numero for numero in numeros if numero % 2 == 0]

print(pares)
#salida: [2,4,6,8,10]
```

También tenemos la posibilidad de **anidar las listas** de comprensión para manejar estructuras más complejas, como listas de listas (matrices).

Ejemplo

```
matriz = [[1,2,3], [4,5,6], [7,8,9]]
agrupada= [elemento for fila in matriz for elemento in fila]

print(agrupada)
#salida: [1,2,3,4,5,6,7,8,9]
```

Las listas por comprensión son una herramienta muy útil de Python que permiten escribir un código más limpio y eficiente.

¿Qué es un argumento en Python?

Un argumento es un valor que se le pasa a la función y que esta puede usar como variable para operar con él, son fundamentales, por tanto, para el funcionamiento de las funciones.

Los argumentos que puede recibir una función deben asignarse al definir esta, situándolos entre los signos de paréntesis; ese nombre actúa como una variable interna de la función. Las funciones pueden tener cero o más argumentos.

Los principales tipos de argumentos son:

- **Argumentos Posicionales**

  Son los argumentos más simples y se pasan a la función en el orden en que se definen en la declaración de la función. Se les asigna un valor por posición, es decir, el primer argumento se asigna al primer parámetro de la función, el segundo argumento al segundo parámetro, y así sucesivamente.

Ejemplo

```
def saludar(nombre):
    print("¡Hola,", nombre, "!")

saludar("Ana")
#Salida: ¡Hola, Ana!
```

- **Argumentos con Nombre** (Keyword Arguments)

  Los argumentos con nombre se pasan a la función utilizando el nombre del parámetro al que se asigna el valor. Esto permite que los argumentos se pasen en cualquier orden, sin importar la posición en que se definieron los parámetros.

Ejemplo

```
def saludar(nombre, saludo):
    print(saludo,nombre)

saludar(nombre= "Ana", saludo= "¡Hola,")
#Salida: ¡Hola, Ana
```

- **Argumentos por Defecto**

  Las funciones pueden tener argumentos con valores por defecto, valores predefinidos. Estos argumentos son opcionales cuando se llama a la función. Si no se proporciona un valor, se utiliza el valor por defecto.

Ejemplo

```
def saludar(nombre, saludo="¡Hola,"):
    print(saludo,nombre)

saludar("Ana")
#Salida: ¡Hola, Ana → Usa el valor por defecto para el saludo
Saludar ("Ana", "¡Egunon,")
#Salida: ¡Egunon, Ana
```

- **Argumentos Arbitrarios**

  En ocasiones, no se sabe de antemano cuántos argumentos se pasarán a una función. En estos casos se pueden usar argumentos arbitrarios. Pueden, a su vez, ser de diversos tipos.

  - Argumentos arbitrarios **posicionales ('*args')**
    Se utilizan para pasar un numero variable de argumentos posicionales a una función.

Ejemplo

```
def sumar(*números):
    resultado= sum(números)
    print("La suma es:", resultado)
sumar(1,2,3) #Salida, la suma es :6
sumar(10,30) #Salida, la suma es :40
```

  - Argumentos arbitrarios **con Nombre ('**kwargs')**
    Se utilizan para pasar un número variable de argumentos con nombre a una función.

Ejemplo

```
def describir_persona(**caracteristicas):
    for clave, valor in caracteristicas.items():
        print(f"{clave}: {valor}")

describir_persona(nombre="Ana", edad=52, ciudad="Ermua")
#Salida:
nombre: Ana
edad: 52
ciudad: Ermua
```

Es posible combinar diferentes tipos de argumentos en una sola función.

El orden correcto es:

1. Argumentos posicionales
2. Argumentos con nombre
3. ***args**
4. ****kwargs**.

Los argumentos en Python son una herramienta fundamental para crear funciones flexibles y reutilizables. Al comprender los diferentes tipos de argumentos y cómo combinarlos, podemos escribir código más eficiente, elegante y fácil de mantener.

¿Qué es una función Lambda en Python?

Una función lambda en Python, también llamada función anónima, es una función simple y de una sola expresión que puede ser utilizada directamente en el lugar donde se define. Estas funciones son útiles para tareas rápidas y de corto plazo donde definir una función completa podría ser innecesario.

Se declaran con la palabra reservada **“ lambda”** de la siguiente manera:

```
Lambda ARGUMENTOS: EXPRESIÓN
```

Donde ARGUMENTOS son los argumentos que admite la función (separados por comas, si hay más de uno) y EXPREXIÓN es la expresión que se evalúa y cuyo resultado será el valor retornado.

Características de las funciones Lambda:

- **Sintaxis concisa:** Las funciones lambda permiten definir funciones en una línea, lo que las hace muy prácticas para su uso en expresiones más complejas.
- **Sin nombre:** A diferencia de las funciones definidas con **“def”,** las funciones lambda no tienen un nombre asociado, de ahí el término “anónimas”.
- **Retorno implícito:** La expresión en una función lambda se evalúa y su valor se devuelve automáticamente, por lo que no es necesario utilizar la palabra clave **“return”.**

Ejemplo

```
def suma(a,b):
    return a+b
print(suma (2,2))
#Salida:4

--- Usando lambda ----
suma = lambda a,b : a + b
print(suma (2,2))
#Salida:4
```

Las funciones lambda se usan comúnmente como argumentos para funciones de orden superior como **“map”**, **“filter”** y **“sorted”.** Estas funciones toman una o más funciones como argumento, y aquí es donde las funciones lambda demuestran su eficacia, ya que

se pueden definir rápidamente en el momento sin necesidad de declarar una función aparte.

Tomemos como ejemplo la función **"map",** que nos permite aplicar una función sobre los ítems de un objeto iterable (lista, tupla..).

Ejemplo

```
def cuadrado(numero):
    return numero * numero
lista = [1,2,3,4,5]
resultado = map(cuadrado,lista)
lista_resultado = list(resultado)
print(lista_resultado)
#Salida: [1, 4, 9, 16, 25]

--- Usando lambda ----
Lista = [1,2,3,4,5]
Resultado= map(lambda numero:numero*numero,lista)
lista_resultado = list(resultado)
print(lista_resultado)
#Salida: [1, 4, 9, 16, 25]
```

En conclusión, las funciones lambda son útiles cuando necesitamos una pequeña función anónima para realizar una tarea específica, en lugar de una función definida que tal vez nunca volveremos a utilizar. S sintaxis es simple y fácil de leer, haciendo que el código sea más legible y mantenible.

¿Qué es un paquete pip?

Un paquete pip es un conjunto de código Python que incluye uno o más módulos o bibliotecas. Estos paquetes generalmente se publican en el índice de paquetes de Python (PyPI) y se pueden instalar fácilmente usando pip, una herramienta de administración de paquetes para Python. Los paquetes de Python pueden contener módulos, subpaquetes y recursos adiciones, como documentación y archivos de datos, facilitando la distribución y reutilización de código.

Pip generalmente se instala de forma predeterminada cuando instalamos Python. Es posible verificar su instalación ejecutando el siguiente comando en la terminal:

Ejemplo

```
pip --versión
#salida:
pip 24.0 from C:\Python312\Lib\site-packages\pip
(python 3.12)
```

o

Si no está instalado se puede descargar el script get-pip.py del sitio web oficial y ejecutarlo para instalar PIP.

Pip es esencial para la creación de software en Python, ya que permiten a los desarrolladores aprovechar el trabajo de otros, reutilizando el código, lo que ahorra tiempo y esfuerzo. Existe una gran cantidad de paquetes disponibles en PyPI, lo que proporciona una amplia gama de funcionalidades para casi cualquier tarea de desarrollo Python.

Para instalar un paquetes adicional usando pip, se utiliza el comando "**pip install"**. Por ejemplo, si deseas instalar Numpy , una biblioteca de Python para computación científica, manejo de datos, puedes hacerlo de la siguiente forma:

Ejemplo

1- Primero asegúrate de tener pip instalado y actualizado:

```
pip install –upgrade pip
```

2- Luego, instala NumPy

```
pip install numpy
```

Una vez instalado ya puedes utilizar Numpy en tu código Python:

```
Import numpy as np
```

En resumen, pip y los paquetes de Python permiten una eficiente gestión y distribución de bibliotecas y módulos, facilitando el desarrollo y la colaboración en proyectos de software.