1 - ¿Qué es un condicional?

Los condicionales son fragmentos de código que dan un resultado dependiendo de la información que se les proporcione, es decir, que dependiendo de qué datos este recibiendo podrá crear diferentes situaciones.

Por ejemplo, tenemos una página web cuyo título en la página principal es un saludo al usuario. Aquí podremos crear una condicional simple y que reciba la información del login del usuario, si el usuario esta logueado se le saludara con su nombre propio, en el caso de que el usuario aun no este logueado o no tenga cuenta, el saludo en vez de contener un nombre propio, llamara al usuario "Invitado"

En el ejemplo propuesto, la condicional sería la siguiente:

- if login == True:
    - print(f"Bienvenido, {nombre_usuario}")
- else:
    - print(f"Bienvenido, invitado")

Paso a detallar el funcionamiento y la sintaxis:

Los condicionales siempre deben empezar con "if" seguido de las condiciones, que en este ejemplo es "login == True" y cerrar la línea con doble punto ":"

Las condiciones son comparaciones entre variables y datos para comprobar si la condición es válida o no lo es, en nuestro caso la condición es "login == True", esto quiere decir que la variable "login" debe ser igual a "True" y para ello se ha usado el doble símbolo de igual "==". No se debe confundir con el símbolo de igual simple "=", este se usa para dar un valor a una variable pero no para hacer una comparación de la variable en sí.

Existen diferentes tipos de comparadores:

- "==": Significa que la variable debe ser idéntica al dato con el que comparar.
- "!=": Significa que la variable debe ser distinta al dato con el que comparar.
- ">": Significa que el valor de la variable debe ser mayor que el dato a comparar.
- "<": Significa que el valor de la variable debe ser menor que el dato a comparar.
- ">=: Significa que el valor de la variable debe ser mayor o igual que el dato a comparar.
- "<=": Significa que el valor de la variable debe ser menor o igual que el dato a comparar.

Si la condición se cumple (Que el usuario ha hecho login) entonces se llevara a cabo lo que haya dentro del "if", para que no de error es importante que las líneas que se vayan a llevar a cabo si la condición es correcta no estén al mismo nivel que el "if" sino que estén con un margen hacia el interior para que Python sepa que ese es el código que se debe ejecutar.

Continuando con la explicación del funcionamiento, ahora debemos rellenar la condicional con lo que ocurrirá en el caso de que el usuario no se haya logueado, que en comparación con el ejemplo, es como si la condición fuera "login == False", para esto se debe escribir al mismo nivel de espacio del "if", un "else" seguido de dos puntos ":", pero esta vez sin añadir ninguna condición ya que la condición principal está en el "if"

La sintaxis es la misma que en la línea del "if", se debe dejar un espacio de margen para que Python sepa que el código pertenece a "else"

En este ejemplo el funcionamiento es muy simple, la condición trabaja con un booleano por lo que solo puede ser True o False, pero para condicionales que funcionan con rangos, por ejemplo con números, existe una tercera sintaxis llamada "elif" y su función es para añadir más niveles al condicional,

Por ejemplo, hay una web de alquiler de vehículos que dependiendo de la edad te dejan alquilar un tipo concreto de vehículos, si se tiene menos de 18 años, no se tiene carnet y no se puede alquilar, desde los 18 años hasta los 24 años, solo se pueden alquilar utilitarios y de 25 años en adelante también se puede alquilar deportivos.

En el ejemplo propuesto, la condicional sería la siguiente:

- if age < 18:
    - print("No tienes carnet de conducir")
- elif age >= 18 and age <= 24:
    - print("Solo se permiten utilitarios")
- else:
    - print("Se puede alquilar utilitarios y deportivos")

Paso a detallar el funcionamiento:

Tal como se aprecia en el ejemplo, usando "elif" he podido añadir una tercera opción ya que al trabajar con edades se puede ampliar las condiciones para tener más posibilidades. La sintaxis es la misma que en el "if", se escribe "elif" seguido de las condiciones y se termina con dos puntos ":". Y al igual que el "if" y el "else" se debe dejar un margen al código que se va a ejecutar para que Python sepa cuál es.

Se pueden usar tantos "elif" como se necesiten, pero solo se puede usar un "if" y "else" por cada condicional, independientemente de la cantidad de condiciones que se deban añadir.

En mi último ejemplo, dentro del "elif" he tenido que usar dos condiciones a la vez, que sea mayor de 18 y que sea menor de 24, paso a detallar cómo funciona el uso de diferentes condiciones.

Para trabajar con diferentes condiciones usamos "and", "or" y "and not"

- "and": Usamos "and" entre dos condiciones cuando queremos que ambas condiciones sean validas, si ninguna o solo una de las condiciones se cumple, entonces no se llevara a cabo el código que este anidado dentro del "if" y se pasara directamente al siguiente "elif" (Si lo hay) o al "else"
    - Por ejemplo, para hacer login en una web se necesita que tanto el correo como la contraseña sean validos de forma obligatoria, si uno de ellos no es correcto, no se podrá hacer login
- "or": Usamos "or" entre dos condiciones cuando necesitamos que al menos una de ellas sea válida. Con que una de las dos sea válida, entonces se ejecutara el "if" y solo se pasara al "elif" o "else" en el caso de que ninguna de las dos condiciones sea válida.
    - Por ejemplo, si deseamos alquilar una moto de baja cilindrada lo podremos hacer si tenemos carnet de moto o carnet de coche, no hace falta que se cumplan las dos condiciones, con que solo una de ellas sea válida ya podrá alquilar la moto de baja cilindrada.
- "and not": Usamos "and not" cuando queremos que la primera condición sea verdadera y la segunda condición sea de forma obligatoria falsa. No hay que confundir con "or", en el caso de "or" puede ser verdadera cualquiera de las dos o las dos directamente, pero con "and not" la primera condición debe ser siempre verdadera y la segunda condición siempre falsa.
    - Por ejemplo, si deseamos alquilar un coche debemos tener carnet de conducir y a la vez NO debemos tener multas por límites de velocidad. La primera condición debe ser siempre verdadera (Tener carnet de conducir) y la segunda condición obligatoriamente debe ser falsa (No tener multas por límites de velocidad)

Si deseamos usar más de dos condiciones debemos encerrarlas entre paréntesis para que el funcionamiento sea correcto

Por ejemplo y usando ejemplos ya descritos, queremos alquilar una moto de baja cilindrada, entonces debemos tener carnet de moto o coche y no tener multas, entonces lo haríamos de la siguiente forma:

- if (carnet_moto == True or carnet_coche == True) and not multas == True:
  - print("Puedes alquilar una moto")
- else:
  - print("No se te permite alquilar una moto")

En este caso lo que va entre paréntesis (Si se tiene carnet) va junto y se considera una condición independiente de lo que esta fuera del paréntesis (Si se tiene multas)

Por último explicare lo que son los operadores ternarios. Sirven para poder comprimir una condicional y de esta forma ahorrarse código y hacerlo más fácil de poder leer.

Usare el mismo ejemplo del login del usuario:

- if login == True:
  - mensaje = f"Bienvenido, {nombre_usuario}"
- else:
  - mensaje = f"Bienvenido, invitado"
- print(mensaje)

Usando operadores ternarios nos quedaría de la siguiente forma:

- mensaje = f"Bienvenido, {nombre_usuario}" if login == True else f"Bienvenido, invitado"
- print(mensaje)

La explicación es la siguiente, debemos visualizar la línea en tres partes, la primera que es hasta el "if" corresponde al código que se debe ejecutar si la condición es verdadera.

La segunda parte se corresponde con el "if" y la condición, que en este caso es si el login es verdadero.

Y la tercera parte es el "else" que se corresponde a lo que se debe añadir a la variable "mensaje" en el caso de que la condición sea falsa.

2 - ¿Cuáles son los diferentes tipos de bucles en python? ¿Por qué son útiles?

Existen dos tipos de bucles: “for in” y “While”

Explicare primero el bucle “for in”

Este bucle sirve para cuando sabemos el número de repeticiones que vamos a tener que hacer basándonos en un numero definido de datos, por ejemplo, si tenemos una lista con 100 datos, sabremos que el el bucle “for in” va a hacer 100 repeticiones, realmente nosotros no necesitamos saber que la lista tiene 100 datos, pueden ser 100 o 100.000, pero si sabemos que sea cual sea el numero de datos que tiene la lista, las repeticiones van a ser una por cada dato y una vez que termine los datos de la lista, entonces el bucle “for in” parara de hacer repeticiones.

El bucle “for in” se puede usar con listas, conjuntos, tuplas y diccionarios, y hará una repetición por cada dato que haya dentro de ellos.

Por ejemplo, tenemos una lista con los números de 1 al 100 y queremos que pase por cada uno de esos números, los multiplique por dos y haga una nueva lista con todos los números ya multiplicados, el bucle sería el siguiente:

```
mi_lista = list(range(1, 101))
nueva_lista = []
for numero in mi_lista:
    numero *= 2
    nueva_lista += [numero]
```

Explicare cada una de las líneas para que se pueda entender mejor:

La primera línea es la creación de una variable tipo lista con los números del 1 al 100 usando los rangos.

La segunda línea es la creación de una variable tipo lista pero vacia, que será la que tendrá los números multiplicados

La tercera línea es el principio del bucle, aquí es donde debemos darle la información sobre los datos con los que va a trabajar, primero se escribe el “for” y un nombre de variable que será la que internamente dentro del bucle usara para los datos individuales, esta variable es únicamente para el uso dentro del bucle, después escribimos “in” y el nombre de la variable de donde son originalmente los datos, que en este caso es “mi_lista” y por ultimo cerrar la línea con dos puntos “:”

Con esto le estamos diciendo que coja todos los datos de la variable “mi_lista”, los meta uno a uno en una nueva variable llamada “numero” y haga una repetición con cada uno de los datos que contiene la lista.

Las siguientes dos líneas son dentro del bucle por lo que se van a repetir tantas veces como datos haya en la variable “mi_lista” y al igual que con las condicionales, debemos dejar un margen para que Python sepa que está dentro del bucle.

Las dos líneas dentro del bucle lo que hacen es primero multiplica cada numero por dos y en la segunda línea mete ese número ya multiplicado dentro de la variable “nueva_lista” que ya creamos previamente fuera del bucle.

Con cada una de las repeticiones multiplica y añade un numero a la nueva lista y se repetirá tantas veces como números haya en la lista original “mi_lista”, una vez que el ultimo numero se haya multiplicado y este dentro de “nueva_lista” el bucle se parara ya que no tiene más datos con los que trabajar.

Dentro del bucle es posible añadir condicionales y otros bucles para hacer operaciones más complejas, por ejemplo, si tenemos una lista con más listas anidadas, necesitamos también anidar un bucle dentro de otro bucle:

- mi_lista = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
- nueva_lista =[]
- for numero_lista in mi_lista:
    - for numero in numero_lista:
        - numero *= 2
        - nueva_lista += [numero]

En este caso tenemos tres listas anidadas dentro de una lista, necesitamos hacer un primer bucle para "desanidar" las listas y después otro bucle para separar los números de cada una de las tres listas y por último la misma operación que en el otro ejemplo, multiplicación por dos y añadir cada número ya multiplicado dentro de una nueva lista.

Ahora explicare el bucle "while"

Este bucle funciona con condiciones, es decir, se ejecutara y hara repeticiones siempre que se cumpla una condición, una vez que la condición se ha cumplido, el bucle terminara. Hay que tener un especial cuidado con este bucle porque si no programamos bien cuando debe parar, podría producir un bucle infinito que hará que el sistema acabe dando fallos.

En el siguiente ejemplo voy a hacer que el bucle "while" imprima en consola todos los números del 1 al 10

- mi_numero = 0
- while mi_numero < 10:
    - mi_numero += 1
    - print(mi_numero)

Paso a explicar el funcionamiento de las líneas:

La primera línea es la creación de una variable con el valor cero para poder usar con el bucle.

La segunda línea es la creación del bucle, escribimos "while" y seguido la condición que debe cumplir para que el bucle continúe la repetición, en este caso le estamos diciendo que continúe haciendo repeticiones mientras la variable "mi_numero" sea más pequeña que 10, es decir, que cuando el bucle detecte que la variable ha llegado a 10, entonces dejara de hacer repeticiones, y por ultimo terminar con dos puntos ":".

Las líneas dentro del bucle deben tener el mismo aspecto que en las condicionales y el bucle "for in", es decir, debe tener un margen para estar dentro del bucle "while"

Las dos líneas dentro del bucle son para sumar una unidad a la variable "mi_numero" y la segunda es para sacar el valor en consola y asi ver todos los números desde el 1 hasta el 10 (Ambos incluidos)

Un ejemplo de bucle infinito usando el bucle de arriba es si la línea "mi_numero = 0" la metemos dentro del bucle encima de "mi_numero += 1". Al hacer esto, aunque se sume una unidad, cuando el bucle hace la siguiente repetición, la variable volverá a estar a cero y nunca se podrá cumplir la condición de parada cuando "mi_numero" llegue a 10.

3 - ¿Qué es una comprensión de listas en python?

Una compresión de listas en Python es cuando creamos una lista nueva y en los datos de la variable añadimos un bucle o condicional, de esta forma podremos ahorrarnos líneas de código y hacer una operación mas rápido.

Por ejemplo, tenemos una lista con números del 1 al 20 y queremos multiplicar por 5 cada número para añadirlos a una nueva lista, de la forma tradicional seria de la siguiente forma:

```
mi_lista = list(range(1, 21))
nueva_lista = []
for numero in mi_lista:
    numero *= 5
    nueva_lista += [numero]
```

Tal como se vio en la pregunta sobre bucles, aquí se crea una lista con un rango de números desde el 1 hasta el 20 (Ambos incluidos) y después se crea una nueva lista vacía

Una vez dentro del bucle multiplicamos cada uno de los números por cinco y lo añadimos a la lista vacia creada anteriormente, con lo que una vez que el bucle finaliza, tendremos todos los números de la lista “mi_lista” multiplicados por 5 y dentro de “nueva_lista”

Ahora creare este mismo ejemplo usando la comprensión de listas que quedaría de la siguiente forma:

```
mi_lista = list(range(1, 21))
nueva_lista = [numero * 5 for numero in mi_lista]
```

Ahora solo tenemos dos líneas en vez de las cinco originales.

La primera línea es la creación de la lista con los números desde el 1 hasta el 20 y en la segunda línea se comprime el resto, voy a detallar:

Comenzamos con la creación de una variable tipo lista y entre los corchetes debemos verlo en dos partes distintas, la primera es la multiplicación de cada número por 5 y la segunda es la cabecera del bucle “for in”

La variable “numero” es la misma creada en el bucle “for in” y deben tener el mismo nombre para evitar errores a la hora de ejecutar el bucle.

Al igual que la forma tradicional de usar bucles “for in” también podremos añadir condicionales. Por ejemplo, usando la misma lista de números del 1 al 20, quiero crear una nueva lista solo con los números que son mayores de 15 (incluido)

De la forma tradicional seria:

```
mi_lista = list(range(1, 21))
nueva_lista = []
for numero in mi_lista:
    if numero >= 15:
        nueva_lista += [numero]
```

Una vez usada la comprensión de listas quedaría de esta otra forma:

```
mi_lista = list(range(1, 21))
nueva_lista = [numero for numero in mi_lista if numero >= 15]
```

Como se puede apreciar todo es igual con la diferencia de que al final de los datos del bucle “for in” añadimos los datos de la condicional, en este caso con la condición de que el número dentro de la variable “numero” sea mayor o igual que 15, en caso contrario el numero en cuestión no se guardara en la variable “nueva_lista”

4 - ¿Qué es un argumento en Python?

Los argumentos son datos que necesitan las funciones en Python para que puedan funcionar, es decir, son los datos con los que trabajan las funciones para que el resultado sea correcto.

Por ejemplo, voy a crear una función de saludo al que se le deben pasar dos argumentos:

- def saludos(nombre, apellido):
  - print(f'Hola, {nombre} {apellido}')
- saludos("Ivan", "Carballo")

Paso a explicar este código:

La primera línea es la creación de la función, la forma de hacerlo es comenzar con "def", seguido del nombre de la función (El que nosotros consideremos apropiado) y terminado entre paréntesis. Dentro de los paréntesis añadimos los argumentos, que en este caso son "nombre" y "apellido", estos dos argumentos son los que la función va a pedir para que pueda ejecutarse de forma correcta, y también son nombres que nosotros decidimos, pero hay que tener en cuenta que tanto el nombre de la función como el de los argumentos deben ser descriptivos ya que serán muy usados y tendremos que aprenderlos. Y por ultimo debemos terminar la línea con dos puntos ":"

La segunda línea que ya está dentro de la función, debe tener un espacio de margen para que Python sepa que esa línea pertenece a la función. Aquí creamos un "print()" y entre los paréntesis escribimos como queremos que sea el saludo, en este caso comenzaremos con un "Hola" y seguido el nombre y apellido, que son los mismos que hemos pasado como argumentos en la función.

Por último debemos ejecutar la función para que entre en funcionamiento. Para ello llamamos a la función "saludos" y entre paréntesis debemos decirle cual es el valor de los dos argumentos, en este caso son dos strings y se escriben en el mismo orden en el que la función pide los argumentos, si se escriben al revés, la función creerá que el nombre es el apellido y estarán los dos datos cruzados.

Para evitar esto lo que se puede hacer es al escribir la línea para ejecutar la función, dar el nombre del argumento a los argumentos como tal para que la función pueda detectarlo y quedaría de la siguiente manera:

- def saludos(nombre, apellido):
  - print(f'Hola, {nombre} {apellido}')
- saludos(nombre = "Ivan", apellido = "Carballo")

De esta forma aunque los pongamos en otro orden, sabrá que valor corresponde a cada argumento

Si queremos que el numero de argumentos sea variable sin necesidad de crear algo complejo, se puede usar "*args" como argumento de la función y cuando llamamos a la función podremos poner tantos datos como queramos.

Usando el ejemplo anterior seria de la siguiente forma:

- def saludos(*args):
  - print(f"Hola, {' '.join(args)}")
- saludos("Ivan", "Carballo", "Machuca")

Como se puede ver, debemos añadir "*args" dentro de los paréntesis al crear la función y después en nuestro "print()" debemos añadir dentro de las llaves "{' '.join(args)}" para que cuando pasemos los valores al llamar a la función los añada todos seguidos y en el mismo orden en el que se han escrito.

En el caso de que alguno de los argumentos queremos decidir sobre donde colocarlo, es posible pasar a la vez un "*args" y argumentos con nombres.

Se haría de la siguiente manera:

- def saludos(hora_dia, *args):
    - print(f"Hola, {' ' .join(args)}, que vaya bien la {hora_dia}")
- saludos(“tarde”, “Ivan”, “Carballo”, “Machuca”)

En este ejemplo hemos añadido un argumento con nombre “hora_dia” y podemos decidir en qué lugar del “print()” colocarlo.

Existe un último argumento que es “**kwargs” que funciona como un diccionario, es decir, trabaja con Keys.

Usando ejemplos anteriores seria de esta forma:

- def saludos(hora_dia, **kwargs):
    - print(f"Hola {kwargs['nombre']} {kwargs['apellido1']} {kwargs['apellido2']} ,que vaya bien la {hora_dia}")
- saludos('tarde', nombre = 'Ivan', apellido1 = 'Carballo', apellido2 = 'Machuca')

Cuando queremos usar este tipo de argumento dentro de la función, la llamada debe ser entre llaves, el argumento “**kwargs” seguido del nombre del resto de argumentos entre corchetes: “{kwargs{‘nombre’]}”

Al igual que con “*args” se pueden poner todos los argumentos que se deseen.

5 - ¿Qué es una función de Python Lambda?

Una función Lambda sirve para poder encerrar en una variable los argumentos que se quieran pasar a una función y de esta forma poder reutilizarse y además no se necesita que aparezca en la función todos los argumentos, pueden ser dinámicos con lo que se puede aprovechar mejor las funciones ya creadas.

Un ejemplo es el siguiente:

- nombre_completo = lambda nombre, apellido1, apellido2: f'{nombre} {apellido1} {apellido2}'

Aquí tenemos una variable llamada “nombre_completo” a la que añadimos lo primero “lambda”, después el nombre de los argumentos y por último la estructura de salida de dichos argumentos, esto por si solo no hace nada, debemos añadirlo a una función junto con la llamada a la propia función, por lo que se vería de la siguiente forma:

- nombre_completo = lambda nombre, apellido1, apellido2: f'{nombre} {apellido1} {apellido2}'
- def saludos(nombre):
    - print(f'hola {nombre}')
- saludos(nombre_completo('Ivan', 'Carballo', 'Machuca'))

Esta función es muy sencilla, imprime en consola un saludo para el usuario.

La función la creamos con un solo argumento “nombre” y después un “print()” con el texto de saludo y el argumento.

La última línea que es la llamada a la función, es la que enlazara la función lambda con la función, primero hacemos una llamada a la función con su nombre “saludos”, entre paréntesis en vez de poner un valor al argumento “nombre” lo que hacemos es llamar a la variable que contiene la función lambda “nombre_completo” y dentro de un nuevo paréntesis, los datos de los tres argumentos que hemos pasado dentro de lambda, con lo que aunque la función “saludos” solo pida un argumento, nosotros usando la función lambda le estamos pasando tres.

Con esto logramos que la función “saludos” sea más dinámica ya que podremos llamarla desde diferentes funciones lambda, cada una con unos datos distintos como argumentos y con posibilidad de reutilizarlas siempre que se necesite.

6 - ¿Qué es un paquete pip?

Un paquete PIP sirve para poder instalar diferentes paquetes de código de Python y poder utilizarlo en el propio código en el que se está trabajando, es decir, en vez de tener que escribir una parte del código por uno mismo, poder instalar ese código de un externo y de esta forma se puede ahorrar mucho tiempo y esfuerzo.

Gracias a PIP no es necesario escribir código desde cero, sino que hay muchas funciones ya existentes que se pueden importar al código propio y poder crear la aplicación o web más rápido y con más estabilidad, ya que es código que funciona correctamente por haberse probado y usado por la comunidad.