1. ¿QUÉ ES UNA CONDICIONAL?

Las sentencias condicionales cumplen con dicha función. Nos ayudan a controlar el flujo de nuestro programa, decidiendo si unas líneas de código se tienen que ejecutar en función de si se cumplen unas condiciones preestablecidas. En concreto, comprueban si una condición es verdadera o falsa para tomar o no cierta acción.

De no ser por las estructuras de control, el código en cualquier lenguaje de programación sería ejecutado secuencialmente hasta terminar. Un código, no deja de ser un conjunto de instrucciones que son ejecutadas unas tras otra. Gracias a las estructuras de control, podemos cambiar el flujo de ejecución de un programa, haciendo que ciertos bloques de código se ejecuten si y solo si se dan unas condiciones particulares.

- **SENTENCIA IF**

La sentencia condicional más básica en Python es la sentencia if, la cual se expresa de la siguiente forma:

```
if condición:
    # ejecutar un código
```

En la expresión previa:

La condición es una expresión booleana que se evalúa como verdadera (True) o falsa (False).

Se requiere el uso de dos puntos (:) al final de la condición.

Todas las líneas de código a ejecutar si se cumple la condición tienen que estar indentadas respecto la sentencia if.

La indentación es una característica que diferencia Python de otros lenguajes de programación, dónde el código a ejecutar de cumplirse la condición se encierra entre llaves. Esta característica tiene el propósito de mejorar la legibilidad de los programas.

Veamos el uso de la sentencia if con un ejemplo:

```
>>> x = 15
>>> if x > 10:
...     print('x es mayor que 10')
...
x es mayor que 10
```

En el ejemplo empezamos asignando a la variable x el valor 15. A continuación evaluamos si cumple la condición de que la variable x es mayor a 10. Como se da el caso, la condición se evalúa como verdadera. Es por ello que el programa ejecuta la sentencia print().

- **SENTENCIA ELSE**

A la sentencia if se le puede añadir opcionalmente una sentencia else. Esta sentencia contiene el código a ejecutar en caso de que no se cumpla la condición de la sentencia if. Esta estructura se expresa del siguiente modo:

```
if condición:
    # ejecutar un código
else:
    # ejecutar un código distinto
```

```
if condición:
    # ejecutar un código
else:
    # ejecutar un código distinto
```

En este caso a la variable x se le asigna el valor 5. Como ahora no es verdad que x sea mayor que 10, la condición de la sentencia if se evalúa como falsa. Por tanto, el programa ejecuta la instrucción bajo la sentencia else.

- **SENTENCIA ELIF**

A una sentencia if else se pueden añadir un número indefinido de condiciones adicionales a verificar. Estas condiciones se definen mediante la sentencia elif, la cual es una abreviación de else if. Ésta se define así:

```
if condición:
    # ejecutar un código
elif otra condición:
    # ejecutar otro código
else:
    # ejecutar un código distinto
```

Siguiendo con el ejemplo de los números, podemos añadir una sentencia elif del siguiente modo:

```
>>> x = 7
>>> if x > 10:
...     print('x es mayor que 10')
... elif x < 10:
...     print('x es menor que 10')
... else:
...     print('x es 10')
...
x es menor que 10
```

En este caso como x tiene asignado el valor 7, la condición de la sentencia elif se evalúa como verdadera. Por este motivo el programa ejecuta la instrucción print() asociada a esta sentencia.

Un punto importante de las sentencias if elif es que una cuando una condición es evaluada como verdadera se ignoran el resto de condiciones. Este hecho se ilustra en el siguiente ejemplo:

En el ejemplo nunca va a evaluarse la condición de la sentencia elif, ya que los valores mayores que 10 también cumplen la condición de la sentencia if (ser mayores que 5). Por consiguiente, la instrucción print asociada al elif es una línea de código inaccesible.

- **CONDICIONAL ANIDADO**

Los condicionales pueden anidarse dentro de otros condicionales. es decir, se pueden colocar dentro de otros condicionales para evaluar condiciones más complejas.

```
numero = 10
if numero > 0:
    print("El número es positivo.")
    if numero % 2 == 0:
        print("El número es par.")
else:
    print("El número es negativo o cero.")
```

En este caso, si número es mayor que 0, se verifica además si número es par.

- **CONDICIONAL CON EXPRESIONES BOOLEANAS**

Las condiciones pueden ser expresiones booleanas complejas que utilicen operadores lógicos como and, or y not.

```
x = 5

y = 10

if x > 0 and y > 0:

   print("Ambos números son positivos.")
```

Aquí, el bloque de código se ejecuta solo si ambas condiciones (x > 0 y y > 0) son verdaderas.

- **USO DE CONDICIONALES EN COMPREHENSIONS**

Python también permite usar condicionales dentro de listas por comprensión. Los condicionales en comprehensions en Python permiten crear listas, conjuntos o diccionarios a partir de iterables existentes de manera concisa y eficiente, con la posibilidad de incluir condiciones para filtrar elementos o modificar valores en función de ciertas condiciones.

Las list comprehensions permiten crear nuevas listas aplicando una expresión a cada elemento de un iterable y opcionalmente filtrando elementos con una condición.

Ejemplo: Filtrar elementos de una lista:

```
numeros = [1, 2, 3, 4, 5]

pares = [num for num in numeros if num % 2 == 0]

print(pares)  # Salida: [2, 4]

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

pares = [num for num in numeros if num % 2 == 0]

print(pares)  # Salida: [2, 4]
```

En este ejemplo, se crea una lista pares que contiene solo los números pares de la lista números.

Transformar y filtrar elementos:

```
cuadrados_pares = [num ** 2 for num in numeros if num % 2 == 0]

print(cuadrados_pares)  # Salida: [4, 16, 36, 64, 100]
```

Aquí se calcula el cuadrado de cada par y se almacena en una lista ‘cuadrados_pares’

El uso de condicionales en comprehensions en Python permite filtrar y transformar datos de manera concisa y legible. Estas técnicas son útiles para:

Crear nuevas estructuras de datos basadas en iterables existentes.

Filtrar elementos según condiciones específicas.

Aplicar transformaciones a elementos de una colección de manera eficiente.

Al usar comprehensions con condicionales, puedes escribir código más compacto y expresivo, mejorando la legibilidad y la eficiencia de tu código.

- **OPERADOR TERNARIO**

El operador ternario o ternary operator es una herramienta muy potente que muchos lenguajes de programación tienen. En Python es un poco distinto a lo que sería en C, pero el concepto es el mismo. Se trata de una cláusula if, else que se define en una sola línea y puede ser usado por ejemplo, dentro de un print().

El operador ternario permite escribir condicionales en una sola línea, Esto es especialmente útil para asignaciones condicionales.

```
edad = 20

mensaje = "Eres mayor de edad" if edad >= 18 else "Eres menor de edad"

print(mensaje)
```

Existen tres partes en un operador ternario, que son exactamente iguales a los que había en un if else. Tenemos la condición a evaluar, el código que se ejecuta si se cumple, y el código que se ejecuta si no se cumple. En este caso, tenemos los tres en la misma línea.

Los condicionales son una herramienta esencial en la programación que permite que los programas tomen decisiones y ejecuten diferentes bloques de código basándose en condiciones específicas. Su uso adecuado puede hacer que el código sea más flexible y adaptable a diferentes situaciones.

2. ¿CUÁLES SON LOS DIFERENTES TIPOS DE BUCLES EN PYTHON? ¿POR QUÉ SON ÚTILES?

En Python, hay dos tipos principales de bucles: el bucle for y el bucle while. Ambos son útiles para automatizar tareas repetitivas, procesar datos, y realizar operaciones sobre estructuras de datos de manera eficiente.

- **BUCLE FOR**

El bucle for se utiliza para iterar sobre una secuencia (como una lista, una tupla, un diccionario, un conjunto o una cadena). Es especialmente útil cuando se sabe de antemano cuántas veces se desea iterar.

El for es un tipo de bucle, parecido al while, pero con ciertas diferencias. La principal es que el número de iteraciones de un for está definido de antemano, mientras que en un while no. La diferencia principal con respecto al while es en la condición. Mientras que en el while la condición era evaluada en cada iteración para decidir si volver a ejecutar o no el código, en el for no existe tal condición, sino un iterable que define las veces que se ejecutará el código. En el siguiente ejemplo vemos un bucle for que se ejecuta 5 veces, y donde la i incrementa su valor “automáticamente” en 1 en cada iteración.

```
for i in range(0, 5):
    print(i)

# Salida:

# 0

# 1

# 2

# 3

# 4
```

Iterables e iteradores

Para entender al cien por cien los bucles for, y como Python fue diseñado como lenguaje de programación, es muy importante entender los conceptos de iterables e iteradores. Empecemos con un par de definiciones:

Los iterables son aquellos objetos que como su nombre indica pueden ser iterados, lo que dicho de otra forma es, que puedan ser indexados. Si piensas en un array (o una list en Python), podemos indexarlo con lista[1] por ejemplo, por lo que sería un iterable.

Los iteradores son objetos que hacen referencia a un elemento, y que tienen un método next que permite hacer referencia al siguiente.

Ambos son conceptos un tanto abstractos y que pueden ser complicados de entender. Veamos unos ejemplos. Como hemos comentado, los iterables son objetos que pueden ser iterados o accedidos con un índice. Algunos ejemplos de iterables en Python son las listas, tuplas, cadenas o diccionarios. Sabiendo esto, lo primero que tenemos que tener claro es que, en un for, lo que va después del in deberá ser siempre un iterable.

Sintaxis:

```
for variable in secuencia:
    # Bloque de código a ejecutar en cada iteración
```

Ejemplo:

Iterar sobre una lista:

```
frutas = ["manzana", "banana", "cereza"]
for fruta in frutas:
    print(fruta)
```

Iterar sobre un rango de números:

```
for i in range(5):
    print(i)
```

Realizar una acción repetida un número específico de veces, como enviar recordatorios de pago.

Iterar sobre un diccionario:

```
estudiante = {"nombre": "Ana", "edad": 21, "carrera": "Ingeniería"}
for clave, valor in estudiante.items():
    print(f"{clave}: {valor}")
```

```
nombre: Ana
edad: 21
carrera: Ingeniería
```

Este es un bucle for que itera sobre cada par clave-valor del diccionario estudiante. El método .items() devuelve una vista del diccionario que contiene tuplas de pares clave-valor. Cada iteración del bucle asigna la clave a la variable clave y el valor a la variable valor.

- **BUCLE WHILE**

El bucle while ejecuta un bloque de código mientras una condición es verdadera. Es útil cuando no se sabe de antemano cuántas veces se necesita iterar y la condición de parada depende de la lógica del programa.

El uso del while nos permite ejecutar una sección de código repetidas veces, de ahí su nombre. El código se ejecutará mientras una condición determinada se cumpla. Cuando se deje de cumplir, se saldrá del bucle y se continuará la ejecución normal. Llamaremos iteración a una ejecución completa del bloque de código.

Cabe destacar que existe dos tipos de bucles, los que tienen un número de iteraciones no definidas, y los que tienen un número de iteraciones definidas. El while estaría dentro del primer tipo.

```
x = 5

while x > 0:

   x -=1

   print(x)

# Salida: 4,3,2,1,0
```

En el ejemplo anterior tenemos un caso sencillo de while. Tenemos una condición x>0 y un bloque de código a ejecutar mientras dure esa condición x-=1 y print(x). Por lo tanto mientras que x sea mayor que 0, se ejecutará el código. Una vez se llega al final, se vuelve a empezar y si la condición se cumple, se ejecuta otra vez. En este caso se entra al bloque de código 5 veces, hasta que, en la sexta, x vale cero y por lo tanto la condición ya no se cumple. Por lo tanto, el while tiene dos partes:

La condición que se tiene que cumplir para que se ejecute el código.

El bloque de código que se ejecutará mientras la condición se cumpla.

Ten cuidado ya que un mal uso del while puede dar lugar a bucles infinitos y problemas. Cierto es que en algún caso tal vez nos interese tener un bucle infinito, pero salvo que estemos seguros de lo que estamos haciendo, hay que tener cuidado. Imaginemos que tenemos un bucle cuya condición siempre se cumple. Por ejemplo, si ponemos True en la condición del while, siempre que se evalúe esa expresión, el resultado será True y se ejecutará el bloque de código. Una vez llegado al final del bloque, se volverá a evaluar la condición, se cumplirá, y vuelta a empezar. No te recomiendo que ejecutes el siguiente código, pero puedes intentarlo.

- **WHILE CONDICIÓN:**

```
   # Bloque de código a ejecutar mientras la condición sea verdadera
```

Ejemplo: Contar hasta que una condición se cumpla.

```
contador = 0

while contador < 5:

   print(contador)

   contador += 1

respuesta = ""

while respuesta != "sí":

   respuesta = input("¿Quieres continuar? (sí/no): ")
```

Aquí se define una variable llamada respuesta y se inicializa con una cadena vacía. Esta variable se usará para almacenar la respuesta del usuario.

Este es un bucle while que continuará ejecutándose mientras la condición respuesta! = "sí" sea verdadera. Esto significa que el bucle seguirá ejecutándose siempre que la variable respuesta no sea igual a "sí".

Ejecución del Código

Durante la ejecución, el programa repetidamente pedirá al usuario que ingrese "sí" o "no" hasta que la respuesta sea "sí".

Primera Iteración:

respuesta es inicializada como una cadena vacía.

El bucle while verifica si respuesta!= "sí", lo cual es verdadero ("" != "sí").

Se ejecuta input("¿Quieres continuar? (sí/no): ") y el programa espera la entrada del usuario.

El usuario ingresa "no".

respuesta ahora es "no".

Segunda Iteración:

El bucle while verifica si respuesta != "sí", lo cual sigue siendo verdadero ("no" != "sí").

Se ejecuta input("¿Quieres continuar? (sí/no): ") y el programa espera la entrada del usuario.

El usuario ingresa "sí".

respuesta ahora es "sí".

Fin del Bucle:

El bucle while verifica si respuesta != "sí", lo cual es falso ("sí" == "sí").

El bucle termina y el programa continúa con la ejecución del código posterior (si lo hay).

- **ELSE Y WHILE**

Algo no muy corriente en otros lenguajes de programación, pero si en Python, es el uso de la cláusula else al final del while. Podemos ver el ejemplo anterior mezclado con el else. La sección de código que se encuentra dentro del else, se ejecutará cuando el bucle termine, pero solo si lo hace “por razones naturales”. Es decir, si el bucle termina porque la condición se deja de cumplir, y no porque se ha hecho uso del break.

```
x = 5
while x > 0:
    x -=1
    print(x) #4,3,2,1,0
else:
    print("El bucle ha finalizado")
```

Podemos ver como si el bucle termina por el break, el print() no se ejecutará. Por lo tanto, se podría decir que si no hay realmente ninguna sentencia break dentro del bucle, tal vez no tenga mucho sentido el uso del else, ya que un bloque de código fuera del bucle cumplirá con la misma funcionalidad.

- **APLICACIONES COMUNES**

Ejecutar bucles hasta que se cumpla una condición específica: Útil cuando no se sabe de antemano cuántas iteraciones serán necesarias.

Implementar menús o interfaces de usuario: Continuar solicitando entrada del usuario hasta que se reciba una entrada válida.

Esperar eventos o condiciones: Como en programación de redes o juegos.

Comparación y Uso Apropiado

Bucle for: Es más apropiado cuando se sabe de antemano el número de iteraciones o cuando se desea iterar sobre una colección de elementos.

Bucle while: Es más apropiado cuando la ejecución depende de una condición que puede cambiar dentro del bucle y no se conoce el número exacto de iteraciones.

¿Por qué son útiles? Los bucles en Python (for y while) son herramientas esenciales para repetir acciones, iterar sobre colecciones de datos y manejar estructuras complejas de forma eficiente. Su uso adecuado permite escribir código más limpio, modular y comprensible, facilitando la realización de tareas repetitivas y el procesamiento de grandes volúmenes de datos.

3. ¿QUÉ ES UNA LISTA DE COMPRENSIÓN EN PYTHON?

List comprehensions

Una de las principales ventajas de Python es que una misma funcionalidad puede ser escrita de maneras muy diferentes, ya que su sintaxis es muy rica en lo que se conoce como expresiones idiomáticas o idiomatic expressions. Las list comprehension o comprensión de listas son una de ellas.

Vayamos al grano, las list comprehension nos permiten crear listas de elementos en una sola línea de código. Por ejemplo, podemos crear una lista con los cuadrados de los primeros 5 números de la siguiente forma

```
cuadrados = [i**2 for i in range(5)]

#[0, 1, 4, 9, 16]
```

De no existir, podríamos hacer lo mismo de la siguiente forma, pero necesitamos alguna que otra línea más de código.

```
cuadrados = []

for i in range(5):

    cuadrados.append(i**2)

#[0, 1, 4, 9, 16]
```

El resultado es el mismo, pero resulta menos claro. Antes de continuar, veamos la sintaxis general de las comprensiones de listas.

```
# lista = [expresión for elemento in iterable]
```

Es decir, por un lado, tenemos el for elemento in iterable, que itera un determinado iterable y “almacena” cada uno de los elementos en elemento como vimos en este otro post sobre el for. Por otro lado, tenemos la expresión, que es lo que será añadido a la lista en cada iteración.

La expresión puede ser una operación como hemos visto anteriormente i**2, pero también puede ser un valor constante. El siguiente ejemplo genera una lista de cinco unos.

```
unos = [1 for i in range(5)]

#[1, 1, 1, 1, 1]
```

La expresión también puede ser una llamada a una función. Se podría escribir el ejemplo anterior del cálculo de cuadrados de la siguiente manera.

```
def eleva_al_2(i):

    return i**2

cuadrados = [eleva_al_2(i) for i in range(5)]

#[0, 1, 4, 9, 16]
```

Como puedes observar, las posibilidades son bastante amplias. Cualquier elemento que sea iterable puede ser usado con las list comprehensions. Anteriormente hemos iterado range() pero podemos hacer lo mismo para una lista. En el siguiente ejemplo vemos como dividir todos los números de una lista entre 10.

```
lista = [10, 20, 30, 40 , 50]

nueva_lista = [i/10 for i in lista]

#[1.0, 2.0, 3.0, 4.0, 5.0]
```

- **AÑADIENDO CONDICIONALES:**

En el apartado anterior hemos visto como modificar todos los elementos de un iterable (como una lista) de diferentes maneras. La primera elevando cada elemento al cuadrado, y la segunda dividiendo cada elemento por diez.

Pero, ¿y si quisiéramos realizar la operación sobre el elemento sólo si una determinada condición se cumple? Pues tenemos buenas noticias, porque es posible añadir un condicional if. La expresión genérica sería la siguiente.

```
# lista = [expresión for elemento in iterable if condición]
```

Por lo tanto, la expresión sólo se aplicará al elemento si se cumple la condición. Veamos un ejemplo con una frase, de la que queremos saber el número de erres que tiene.

```
frase = "El perro de san roque no tiene rabo"

erres = [i for i in frase if i == 'r']

#['r', 'r', 'r', 'r']
```

Lo que hace el código anterior es iterar cada letra de la frase, y si es una r, se añade a la lista. De esta manera el resultado es una lista con tantas letras r como la frase original tiene, y podemos calcular las veces que se repite con len().

```
print(len(erres))

#4
```

Sets comprehension

Las set comprehensions son muy similares a las listas que hemos visto con anterioridad. La única diferencia es que debemos cambiar el () por {}. Como resulta evidente, dado que los set no permiten duplicados, si intentamos añadir un elemento que ya existe en el set, simplemente no se añadirá.

```
frase = "El perro de san roque no tiene rabo"

mi_set = {i for i in frase if i == "r"}

#{'r'}
```

Dictionary comprehension

Y por último, también tenemos las comprensiones de diccionarios. Son muy similares a las anteriores, con la única diferencia que debemos especificar la key o llave. Veamos un ejemplo.

```
lista1 = ['nombre', 'edad', 'región']

lista2 = ['Pelayo', 30, 'Asturias']

mi_dict = {i:j for i,j in zip(lista1, lista2)}

#{'nombre': 'Pelayo', 'edad': 30, 'región': 'Asturias'}
```

Como se puede ver, usando: asignamos un valor a una llave. Hemos usado también zip(), que nos permite iterar dos listas paralelamente. Por lo tanto, en este ejemplo estamos convirtiendo dos listas a un diccionario.

4. ¿QUÉ ES UN ARGUMENTO EN PYTHON?

En Python, un argumento es un valor que se pasa a una función cuando se llama a esa función. Los argumentos permiten que las funciones reciban datos de entrada y realicen operaciones utilizando esos datos.

Vamos a desglosar este concepto paso a paso.

- **DEFINICIÓN DE UNA FUNCIÓN**

Primero, definimos una función en Python usando la palabra clave def. Una función puede aceptar uno o más argumentos. Aquí hay un ejemplo básico de una función sin argumentos:

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

Para llamar a esta función, simplemente escribimos su nombre:

```
saludar()
```

Esto imprimirá:

```
¡Hola!
```

- **FUNCIONES CON ARGUMENTOS**

Las funciones pueden aceptar argumentos para realizar tareas con datos específicos. Aquí hay un ejemplo de una función con un argumento:

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

Para llamar a esta función, pasamos un argumento (por ejemplo, un nombre):

```
saludar("Ana")
```

Esto imprimirá:

```
¡Hola, Ana!
```

- **MÚLTIPLES ARGUMENTOS**

Una función también puede aceptar múltiples argumentos. Por ejemplo:

```
def sumar(a, b):
    return a + b
```

Para llamar a esta función, pasamos dos argumentos:

```
resultado = sumar(3, 5)
print(resultado)
```

```
8
```

- **ARGUMENTOS CON VALORES POR DEFECTO**

Podemos definir valores por defecto para los argumentos. Si no se proporciona un valor, se usará el valor por defecto:

```
def saludar(nombre="amigo"):
    print(f"¡Hola, {nombre}!")
```

Podemos llamar a esta función con o sin un argumento:

```
saludar()        # ¡Hola, amigo!
```

```
saludar("Luis")    # ¡Hola, Luis!
```

- **ARGUMENTOS DE PALABRA CLAVE (KEYWORD ARGUMENTS)**

Cuando llamamos a una función, podemos especificar los argumentos por su nombre, lo que se conoce como argumentos de palabra clave:

```
def presentar(nombre, edad):
    print(f"Me llamo {nombre} y tengo {edad} años.")
presentar(nombre="Carlos", edad=30)
presentar(edad=25, nombre="María")
```

Ambas llamadas imprimirán:

Me llamo Carlos y tengo 30 años.

Me llamo María y tengo 25 años.

- **ARGUMENTOS ARBITRARIOS**

A veces, no sabemos de antemano cuántos argumentos se pasarán a una función. Podemos usar *args para aceptar un número arbitrario de argumentos:

```
def listar_amigos(*amigos):
    for amigo in amigos:
        print(amigo)
listar_amigos("Ana", "Luis", "Pedro")
```

Esto imprimirá:

Ana

Luis

Pedro

- **ARGUMENTOS DE PALABRA CLAVE ARBITRARIOS**

Podemos usar **kwargs para aceptar un número arbitrario de argumentos de palabra clave:

```
def describir_persona(**kwargs):
    for clave, valor en kwargs.items():
        print(f"{clave}: {valor}")
describir_persona(nombre="Laura", edad=27, ciudad="Madrid")
```

Esto imprimirá:

nombre: Laura

edad: 27

ciudad: Madrid

Argumentos: valores pasados a las funciones para que realicen tareas específicas.

Múltiples argumentos: funciones pueden aceptar varios valores.

Valores por defecto: permiten omitir argumentos opcionales.

Argumentos de palabra clave: especificar valores por su nombre.

Argumentos arbitrarios: *args y **kwargs permiten recibir un número variable de argumentos.

Usar argumentos en funciones es una manera esencial de hacer que nuestro código sea más flexible y reutilizable.

5. ¿QUÉ ES UNA FUNCIÓN LAMBDA EN PYTHON?

Las funciones Lambda son funciones anónimas que solo pueden contener una expresión, son una manera compacta de escribir una función si solo necesitas una expresión corta. Una función lambda en Python es una forma concisa de definir una función anónima, es decir, una función que no tiene un nombre explícito. Estas funciones son útiles para operaciones sencillas que se pueden definir en una sola línea de código. La sintaxis básica de una función lambda es:

¿Para qué sirven las funciones lambda de Python?

Las lambdas surgen de la programación funcional. En algunos lenguajes, como JavaScript, a menudo se utilizan funciones anónimas sin una palabra clave especial. En Python, las expresiones lambda se utilizan para crear pequeñas funciones localmente, sin grandes complicaciones.

Ejemplo:

lambda argumentos: expresión

Aquí hay un desglose de su estructura:

lambda es la palabra clave que indica que se está definiendo una función lambda.

argumentos es una lista de parámetros que la función toma (similares a los argumentos en una función normal).

expresión es una única expresión que la función evalúa y devuelve.

Las funciones lambda son comunes en situaciones donde se necesita una función simple y de corta duración, como argumentos para funciones de orden superior (por ejemplo, map(), filter(), y sorted()).

Nota: En Python, el término “función lambda”se refiere a una función creada con la palabra clave lambda. Lambda no es el nombre de una función y tampoco es uno de los operadores de Python.

Así creas una función en Python:

```
def mi_func(argumentos):
  # cuerpo de la función
```

Se declara la función con la palabra clave def, les das un nombre y entre paréntesis los argumentos que recibirá la función. Puede haber tantas líneas de código como quieras con todas las expresiones y declaraciones que necesites.

Pero algunas veces solo necesitarás una expresión dentro de tu función, por ejemplo, una función que duplique el valor de un argumento:

```
def doble(x):
  return x * 2
```

Esta función puede ser usada dentro de la función map, de la siguiente forma:

```
def doble(x):
  return x * 2
mi_lista = [1, 2, 3, 4, 5, 6]
lista_nueva = list(map(doble, mi_lista))
print(lista_nueva) # [2, 4, 6, 8, 10, 12]
```

Este es un buen lugar para usar una función lambda, ya que puede ser creada en el mismo lugar donde se necesite, esto significa menos líneas de código y así también evitarás nombrar una función que solo utilizaras una sola vez y que tendría que ser almacenada en memoria,

Como usar funciones lambda en Python

Una función lambda se usa cuando necesitas una función sencilla y de rápido acceso: por ejemplo, como argumento de una función de orden mayor como los son map o filter

La sintaxis de una función lambda es lambda args: expresión. Primero escribes la palabra clave lambda, dejas un espacio, después los argumentos que necesites separados por coma, dos puntos :, y por último la expresión que será el cuerpo de la función.

Recuerda que no puedes darle un nombre a una función lambda, ya que estas son anónimas (sin nombre) por definición.

Una función lambda puede tener tantos argumento como necesites, pero debe tener una sola expresión.

Ejemplo 1

Por ejemplo, puedes escribir una función lambda que duplique sus argumentos lambda x: x * 2 y usarla con la función map para duplicar todos los elementos de una lista:

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

lista_nueva = list(map(lambda x: x * 2, mi_lista))

print(lista_nueva)  # [2, 4, 6, 8, 10, 12]
```

Así luce sin una función lambda

```
def doble(x):
  return x*2

mi_lista = [1, 2, 3, 4, 5, 6]

lista_nueva = list(map(doble, mi_lista))

print(lista_nueva) # [2, 4, 6, 8, 10, 12]
```

Puedes notar la diferencia, entre usar una función lambda y el ejemplo con la función doble, ahora el código es más compacto y no hay una función extra utilizando espacio en memoria.

Ejemplo 2

También puedes escribir una función lambda que revise si un número es positivo, lambda x: x > 0, y usarla con la función filter para crear una lista de números positivos.

```
mi_lista = [18, -3, 5, 0, -1, 12]

lista_nueva = list(filter(lambda x: x > 0, mi_lista))

print(lista_nueva) # [18, 5, 12]
```

Una función lambda se define donde se usa, de esta manera no hay una función extra utilizando espacio en memoria.

Si una función es utilizada una sola vez, lo mejor es usar una función lambda para evitar código innecesario y desorganizado.

Ejemplo 3

También es posible que una función devuelva una función lambda.

Si necesitas funciones que multipliquen diferentes números, por ejemplo, duplicar, triplicar, etc... una función lambda puede ser útil

En lugar de crear múltiples funciones, puedes crear una sola función multiplicar_por() y llamarla con diferentes argumentos para crear una función que duplique o triplique.

```
def multiplicar_por (n):
  return lambda x: x * n

duplicar = multiplicar_por(2)

triplicar = multiplicar_por(3)

diez_veces = multiplicar_por(10)
```

La función lambda toma el valor de n de la función multiplicar_por(n) así que en duplicar el valor de n es 2, en triplicar n vale 3 y en diez_veces vale 10. Y al llamar a estas funciones con un argumento podemos retornar el número multiplicado.

```
duplicar(6)

> 12

triplicar(5)

> 15

diez_veces(12)

> 120
```

Si no usáramos funciones lambda, tendríamos que crear una función diferente dentro de multiplicar_por, y se vería algo así:

```
def multiplicar_por(x):
  def temp (n):
    return x*n
  return temp
```

Usando una función lambda el código necesita menos líneas de código y es más legible.

¿Cuál es la diferencia entre lambda y def?

Al principio parece extraño que Python reconozca dos formas de crear funciones: lambda y def. Pero hay que tener en cuenta que lambda no es una función propia, sino simplemente una notación diferente para crear funciones cortas localmente. Todas las funciones creadas con lambda pueden crearse también mediante def. Sin embargo, no ocurre lo mismo a la inversa.

A nivel sintáctico, tanto lambda como def son palabras clave. Una diferencia entre ambas radica en la estricta separación que hace Python entre sentencias (“Statement”) y expresiones (“Expression”). Las sentencias son pasos en la ejecución del código, mientras que las expresiones se evalúan frente a un valor.

Con def comienza una sentencia (concretamente un “Compound Statement”) que contiene más sentencias. Dentro de la sentencia def,y solo allí, pueden aparecer sentencias return. Tras la llamada a la función definida con def, una sentencia return devuelve un valor.

A diferencia de la sentencia def, lambda inicia una expresión que no debe contener ninguna sentencia. La expresión lambda toma uno o más argumentos y devuelve una función anónima. Si se llama a la función lambda generada, se hará una evaluación de la expresión contenida con los argumentos pasados y se devolverá el resultado.

¿Cuáles son las limitaciones de las expresiones lambda de Python?

Python limita específicamente la utilidad de las funciones lambda, ya que suele ser mejor nombrar las funciones. Esto obliga a los programadores a pensar en el sentido de la función y a distinguir claramente unas partes de otras.

A diferencia del cuerpo de una función definida mediante la palabra clave def, las lambdas no pueden contener sentencias. Por tanto, no es posible utilizar if, for, etc. dentro de una función lambda. Tampoco es posible lanzar una excepción (Exception), ya que para ello se necesita la sentencia raise.

Las funciones lambda en Python solo pueden contener una única expresión que se evalúa después de llamarla. Dentro de la expresión lambda, no puedes utilizar anotaciones de tipo. Hoy en día se utilizan otras técnicas para la mayoría de los casos de uso de las funciones lambda de Python. Especialmente en las comprensiones.

6. ¿QUÉ ES UN PAQUETE PIP?

Pip (administrador de paquetes) pip es un sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python. Muchos paquetes pueden ser encontrados en el Python Package Index (PyPI). Python 2.7.9 y posteriores (en la serie Python2), Python 3.4 y posteriores incluyen pip (pip3 para Python3) por defecto. Pip es un acrónimo recursivo que se puede interpretar como Pip Instalador de Paquetes o Pip Instalador de Python

PIP (Pip Installs Packages) es una herramienta de administración de paquetes en Python que permite instalar y gestionar bibliotecas y dependencias adicionales que no forman parte de la biblioteca estándar de Python. Utilizando pip, los desarrolladores pueden descargar e instalar paquetes desde el Python Package Index (PyPI), que es el repositorio oficial de software para Python.

Una ventaja importante de pip es la facilidad de su interfaz de línea de comandos, el cual permite instalar paquetes de software de Python fácilmente desde solo una orden: pip install nombre-paquete

Pip es una herramienta indispensable para cualquier desarrollador de Python, ya que facilita la gestión de paquetes y permite a los desarrolladores aprovechar el trabajo de otros para construir sus propios proyectos. Además, pip hace que sea fácil compartir código y colaborar con otros desarrolladores en la comunidad de Python. Características Principales de pip:

Instalación de Paquetes:

Pip facilita la instalación de paquetes de PyPI mediante un simple comando en la terminal. Por ejemplo:

pip install nombre_del_paquete

Gestión de Dependencias:

Pip se encarga de resolver y descargar todas las dependencias necesarias para un paquete específico. Esto significa que si un paquete A depende de otros paquetes B y C, pip descargará e instalará B y C automáticamente.

Actualización de Paquetes:

Es posible actualizar los paquetes instalados a sus últimas versiones utilizando el comando

pip install --upgrade nombre_del_paquete

Desinstalación de Paquetes:

Pip también permite desinstalar paquetes que ya no se necesitan:

pip uninstall nombre_del_paquete

Listar Paquetes Instalados:

Para ver una lista de todos los paquetes instalados en el entorno actual, se puede usar:

pip list

Archivos de Requisitos:

Los desarrolladores pueden listar las dependencias de su proyecto en un archivo requirements.txt, lo que facilita la instalación de todas las dependencias de un proyecto con un solo comando:

pip install -r requirements.txt

Ejemplo Práctico:

Instalar un Paquete:

Para instalar requests, una popular biblioteca para hacer solicitudes HTTP:

pip install requests

Crear un Archivo de Requisitos:

En el proyecto, crear un archivo requirements.txt con el siguiente contenido:

requests

numpy

Luego, para instalar todos los paquetes listados en el archivo requirements.txt:

pip install -r requirements.txt

Instalación de pip:

En la mayoría de las distribuciones de Python recientes, pip ya está incluido. Sin embargo, si necesitas instalarlo manualmente, puedes usar el siguiente comando (en sistemas Unix):

curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py

python get-pip.py

Para instalar pip en Windows, puedes seguir estos pasos:

Paso 1: Verificar si Python está instalado

Primero, asegúrate de tener Python instalado en tu sistema. Puedes verificar esto abriendo el símbolo del sistema (CMD) y ejecutando el siguiente comando:

python --version

Si Python está instalado, verás la versión de Python. Si no está instalado, puedes descargar el instalador de Python desde python.org y seguir las instrucciones de instalación. Durante la instalación, asegúrate de marcar la opción "Add Python to PATH".

Paso 2: Descargar el script de instalación de pip

Abre tu navegador web y descarga el script get-pip.py desde el siguiente enlace: get-pip.py.

Paso 3: Ejecutar el script de instalación de pip

Abrir el Símbolo del Sistema:

Abre el símbolo del sistema (CMD). Puedes hacerlo buscándolo en el menú de inicio o presionando Win + R, escribiendo cmd y presionando Enter.

Navegar al directorio donde descargaste get-pip.py:

Usa el comando cd para cambiar al directorio donde descargaste el archivo. Por ejemplo, si el archivo está en la carpeta Descargas de tu usuario, ejecuta:

cd C:\Users\TuUsuario\Descargas

Ejecutar el script get-pip.py:

Ejecuta el siguiente comando para instalar pip:

python get-pip.py

Paso 4: Verificar la instalación de pip

Después de que el script se ejecute y termine la instalación, puedes verificar que pip se haya instalado correctamente ejecutando:

pip --version

Deberías ver algo como esto, indicando que pip está instalado y funcionando:

pip 21.0.1 from ... (python 3.x)

Paso 5: Añadir Python y pip al PATH (si no se hizo durante la instalación)

Si el comando pip no se reconoce, es posible que necesites añadir Python y pip al PATH del sistema.

Abrir las Propiedades del Sistema:

Haz clic derecho en "Este PC" o "Mi PC" en el escritorio o en el explorador de archivos.

Selecciona "Propiedades".

Haz clic en "Configuración avanzada del sistema".

En la pestaña "Avanzado", haz clic en "Variables de entorno".

Editar la Variable PATH:

En las "Variables del sistema", encuentra la variable Path y selecciónala.

Haz clic en "Editar".

Añade las rutas a Python y pip (usualmente algo como C:\Python39 y C:\Python39\Scripts).

Haz clic en "Aceptar" en todas las ventanas para aplicar los cambios.

Después de hacer esto, cierra y vuelve a abrir el símbolo del sistema y prueba de nuevo el comando pip --version.

Paso 6: Instalar Paquetes con pip

Una vez que pip esté instalado, puedes usarlo para instalar paquetes de Python. Por ejemplo, para instalar la biblioteca requests, ejecuta:

pip install requests

Pero Pip hace más que simplemente instalar paquetes. También nos permite administrar nuestras dependencias. Por ejemplo, si estamos trabajando en un proyecto que depende de NumPy y SciPy, podemos agregar ambos paquetes a un archivo requirements.txt. Luego, si alguien más clona nuestro proyecto, puede simplemente instalar todas las dependencias del mismo tiempo escribiendo pip install -r requirements.txt.

Es importante tener en cuenta que Pip no siempre es perfecto. Puede haber ocasiones en las que dos paquetes de software diferentes necesiten versiones diferentes de la misma dependencia, lo que puede causar conflictos. En esos casos, la mejor opción es utilizar un ambiente virtual de Python y especificar las versiones de los paquetes que necesitamos para cada proyecto. De esta manera, podemos mantener nuestros proyectos separados y aislados de otros paquetes y dependencias que no necesitamos.

Una práctica común al utilizar Pip es crear un archivo requirements.txt para tener un registro de todas las dependencias necesarias y sus versiones

Al desarrollar un proyecto en Python, es muy común hacer uso de paquetes y librerías externas para aprovechar las funcionalidades que estos nos ofrecen. Sin embargo, esto puede generar un problema de gestión de dependencias, especialmente cuando se trabaja en equipo o se cambia el ambiente de trabajo.

Es por eso que una práctica muy común al utilizar Pip, el gestor de paquetes de Python, es crear un archivo requirements.txt que contenga todas las dependencias necesarias y sus versiones correspondientes. De esta manera, al

instalar todas las dependencias desde este archivo, se asegura que todos los desarrolladores en el equipo están trabajando con las mismas versiones de las librerías.

Para crear un archivo requirements.txt, basta con utilizar el siguiente comando en la terminal de comandos de Python:

pip freeze > requirements.txt

Este comando genera un archivo llamado requirements.txt en el directorio actual, el cual contiene todas las dependencias instaladas en el ambiente virtual actual y sus versiones correspondientes. Es importante mencionar que este archivo debe ser incluido en el control de versiones del proyecto, de manera que todos los desarrolladores puedan acceder a él.

Una vez que se tiene el archivo requirements.txt, es posible instalar todas las dependencias del proyecto utilizando el siguiente comando:

pip install -r requirements.txt

Este comando nos permite instalar todas las dependencias listadas en el archivo requirements.txt de una sola vez, asegurando que todos los desarrolladores están utilizando las mismas versiones de las librerías.

En resumen, el uso de Pip y la creación de un archivo requirements.txt nos permite gestionar de manera eficiente las dependencias de nuestro proyecto en Python, asegurando que todos los desarrolladores están trabajando con las mismas versiones de las librerías. Es una práctica muy útil para cualquier proyecto de software en Python y es recomendable incluirla en el flujo de trabajo del equipo de desarrollo.