¿Qué es un condicional?

Un condicional es una estructura de control en programación que permite ejecutar ciertas acciones basadas en una condición o conjunto de condiciones. Básicamente, un condicional evalúa una expresión booleana (es decir, una expresión que puede ser verdadera o falsa) y ejecuta un bloque de código dependiendo del resultado de esta evaluación.
Los condicionales son instrucciones que se le dan a un programa para que ejecute un bloque de código u otro si se cumplen determinados requisitos. De este modo, puede adaptarse a distintas situaciones e incluso responder ante usos o consultas inusuales.

El software entiende cada condición como verdadera o falsa, y actúa según dicha evaluación. Para explicar las circunstancias que se valoran, se hace uso de los operadores de Python, especialmente de los lógicos–cuando hay que comprobar más de un requisito a la vez–, y los de comparación –cuando la condición es única–.

Los condicionales suelen usarse junto a otros elementos, como los

operadores lógicos y los de comparación

Gracias a los condicionales que hay en Python, las plataformas pueden ajustarse a la interacción del/la usuario/a,controlar que cumple los requerimientos para registrarse – la longitud de una contraseña, por ejemplo– o que los datos de un/a cliente/a potencial son válidos y pueden entrar en el base de datos como lead. –¿el teléfono tiene la cantidad de números correcta?, ¿y el código postal?–. Para las operaciones más complicadas, se pueden combinar varios condicionales, como veremos en los siguientes epígrafes.

Uso de if, else y elif en Python

Los principales condicionales en Python son **if, else y elif**.
“Traducidos” al lenguaje humano, serían:

Condicionales principales		
Condicional	*Significado*	ejemplo
if	si	edad = 18 if edad < 18: print(“No puedes beber alcohol”.)

else	“si no”	edad = 18
		if edad < 18:
		
		print(“No puedes beber alcohol”.)
		
		else:
		
		print(“Puedes beber alcohol”.)

elif (else + if)	“Si no, si”	edad = 18
		if edad > 18:
		
		print(“Puedes beber alcohol”.)
		
		elif edad == 18:
		
		print(“Como acabas de cumplir la mayoría de edad, puedes beber alcohol”.)
		
		else:
		
		print(“No puedes beber alcohol”.)

Como se deduce a partir de la tabla anterior, cada uno de los condicionales en Python **marca una ruta a seguir:**

- **if:** ejecuta el código al que va asociado si la condición es verdadera. En el ejemplo, sería mostrar el mensaje “No puedes beber alcohol” si el/la usuario/a tiene menos de 18 años.
- **Else:** va después de un if. Aplica el código al que está conectado si la condición que el if marca es falsa. Es decir, si A no se cumple, else B.
- **elif:** si hay varias condiciones que comprobar antes de “decidir”, se utiliza if en la primera y elif en las siguientes. En la tabla, expresa que, si el/la usuario/a no es mayor de 18 años, sino que tiene justo los 18, puede beber alcohol legalmente.

Condicionales anidados y operadores lógicos en Python

Los condicionales anidados en Python son el resultado de**introducir un condicional dentro de otro**. Como las matrioshkas, pero con requisitos en lugar de muñecas.

Como ocurre con el elif, se emplean para**c omprobar distintas variables**. Entonces, ¿qué los distingue exactamente de esta secuencia?

Diferencia entre elif y los condicionales anidados

elif	*Condicionales anidados*
• Solo valora el requisito al que va asociado si los anteriores resultan ser falsos	• Valoran cada requisito independientemente del resto
• Simplifica el código y lo hace más fácil de leer y de comprender	• Permiten mayor precisión en las casuísticas complicadas

Otra de las dudas que pueden surgirte es si los condicionales anidados suponen una **alternativa válida al operador** or. Lo cierto es que ambos hacen lo mismo: determinar si, como mínimo, uno de los requisitos expresados es verdadero. Por ejemplo, estos dos códigos indican las mismas instrucciones:

Condicionales anidados

```
x= 10
if x > 5:
  if x < 15:
    print(“x está entre 5 y 15.”)
  else:
    print(“x es mayor que 15.”)
else:
  print(“x es menor o igual a 5.”)
```

Operador or

```
x = 10
if x > 5 or x > 15:
  if x < 15:
    print(“x está entre 5 y 15.”)
  else:
    print(“x es mayor que 15.”)
else:
  print(“x es menor o igual a 5.”)
```

Hay otros operadores lógicos que pueden sustituir las anidaciones condicionales, como el **and**. Emplear unos y otros depende del grado de detalle que quieras darle al código y de lo **complejidad de las condiciones** que haya que describir. ¡Te recomendamos **practicar con**

ejercicios de Python para ir puliendo tu capacidad de decisión entre unos y otros!

Prácticas recomendadas y errores comunes en el uso de condicionales

Para **usar correctamente los condicionales** en Python hay que seguir las buenas prácticas que recomiendan los/as desarrolladores/as. Estas son:

- Explicar los condicionales más complejos con **comentarios**, para que otros/as programadores/as puedan comprenderlo.
- Colocar las condiciones en la **secuencia correcta**, sobre todo si se usan sentencias como el elif, que valora los requisitos según el orden en el que se hayan escrito.
- Optar siempre por la **opción más sencilla.**Por ejemplo, usar el and cuando se pueda, en lugar de varios if anidados.
- **Recurrir a los paréntesis** cuando se combinan distintos condicionales y operadores.

Pese a que no seguir estos criterios es ya de por sí un **error**, al usar los condicionales en Python suelen cometerse otros como:

- **No escribir los dos puntos “:”** después de los condicionales.
- **Abusar de las anidaciones**, que hacen que el código sea cada vez más complejo y que el software esté menos optimizado.
- **No contemplar todos los requisitos** posibles.
- Realizar una **indentación incorrecta** o confundir los operadores.

Utilidad de los bucles en Python:

Automatización de tareas repetitivas: Los bucles permiten que se ejecute un bloque de código repetidamente, lo que es útil para automatizar tareas repetitivas en un programa.

Procesamiento de datos: Los bucles son esenciales para iterar sobre estructuras de datos como listas, tuplas y diccionarios para procesar o manipular los datos contenidos en ellas.

Implementación de algoritmos: Muchos algoritmos requieren la ejecución repetida de ciertas operaciones sobre datos. Los bucles proporcionan la estructura necesaria para implementar estos algoritmos de manera eficiente.

Interacción con el usuario: Los bucles pueden utilizarse para solicitar entrada del usuario repetidamente hasta que se proporcione una entrada válida o para mantener una interfaz de usuario en funcionamiento hasta que el usuario decida salir.

Control de flujo: Los bucles permiten controlar el flujo de ejecución de un programa, permitiendo la ejecución de diferentes bloques de código dependiendo de ciertas condiciones.

En resumen, los bucles son esenciales en Python (y en la programación en general) porque permiten que los programas realicen tareas repetitivas de manera eficiente y flexible, lo que facilita la automatización, el procesamiento de datos y la implementación de algoritmos, entre otras cosas.

1. Bucle for:

El bucle for se utiliza para iterar sobre una secuencia de elementos, como una lista, tupla, rango, u otro tipo de objeto iterable. Cada elemento de la secuencia se asigna a una variable en cada iteración, y el bloque de código dentro del bucle se ejecuta una vez para cada elemento de la secuencia.

Ejemplo:

```
# Iterar sobre una lista
frutas = ["manzana", "banana", "cereza"]
for fruta in frutas:
    print(fruta)
```

En este ejemplo, el bucle for itera sobre cada elemento de la lista frutas, imprimiendo cada fruta en una línea separada.

2. Bucle while:

El bucle while se utiliza para ejecutar un bloque de código mientras una condición específica sea verdadera. La condición se evalúa antes de cada iteración y, si es verdadera, se ejecuta el bloque de código. El bucle continuará ejecutándose hasta que la condición se vuelva falsa.

Ejemplo:

```
# Imprimir números del 1 al 5 usando un bucle while
numero = 1
while numero <= 5:
    print(numero)
    numero += 1
```

En este ejemplo, el bucle while se ejecuta mientras numero sea menor o igual a 5. En cada iteración, se imprime el valor actual de numero y luego se incrementa en 1. El bucle termina cuando numero es mayor que 5.

Utilidad de los Bucles:

Iteración sobre estructuras de datos: Los bucles permiten recorrer y procesar elementos de listas, tuplas, diccionarios y otros tipos de datos de manera eficiente.

Control de flujo: Los bucles permiten que un programa ejecute ciertas operaciones repetidamente hasta que se cumpla una condición específica.

Automatización de tareas repetitivas: Los bucles son útiles para ejecutar tareas repetitivas sin tener que escribir el

mismo código una y otra vez.

Implementación de algoritmos: Muchos algoritmos requieren la ejecución repetida de ciertas operaciones sobre datos. Los bucles proporcionan la estructura necesaria para implementar estos algoritmos de manera eficiente.

En resumen, los bucles en Python son fundamentales para realizar tareas repetitivas, procesar datos y controlar el flujo de ejecución de un programa. Su flexibilidad y poder los convierten en una herramienta indispensable en la programación.

¿Qué es una lista por comprensión en Python?

Una lista por comprensión en Python es una forma concisa y elegante de crear listas. Permite construir listas de manera eficiente y legible, utilizando una sintaxis compacta que combina un bucle for con una expresión que define los elementos de la lista.
La sintaxis general de una lista por comprensión es la siguiente:

```
nueva_lista = [expresion for elemento in secuencia]
```

Donde:

expresion es la expresión que define los elementos de la nueva lista.
elemento es la variable que representa cada elemento de la secuencia.
secuencia es la secuencia de elementos sobre la cual se itera.

Por ejemplo, considera el siguiente código que crea una lista de los cuadrados de los números del 0 al 9 utilizando un bucle for estándar:

```
cuadrados = []
for i in range(10):
    cuadrados.append(i**2)
```

Con una lista por comprensión, esto se puede expresar de manera mucho más concisa:

```
cuadrados = [i**2 for i in range(10)]
```

Este código hace lo mismo que el bucle for anterior, pero en una sola línea. Cada elemento de la lista cuadrados es el cuadrado de un número del 0 al 9.

Ventajas de las Listas por Comprensión:

Sintaxis Concisa: Las listas por comprensión permiten escribir código de manera más compacta y legible en comparación con el uso de bucles for tradicionales.

Eficiencia: Las listas por comprensión suelen ser más

eficientes en términos de tiempo de ejecución que los bucles for estándar, especialmente en listas grandes.

Expresividad: Ayudan a expresar la intención del código de manera clara y directa, lo que facilita la comprensión del código por parte de otros programadores.

En resumen, las listas por comprensión son una característica poderosa de Python que permite crear listas de forma concisa y elegante, lo que resulta en un código más claro y eficiente. Son una herramienta fundamental en el kit de herramientas de cualquier programador de Python.

¿Qué es un argumento en Python?

En Python, un argumento se refiere a un valor que se pasa a una función o método cuando se llama. Los argumentos son utilizados para proporcionar datos a la función para que pueda realizar su tarea. En una definición de función, los argumentos son los valores que se colocan entre paréntesis y se utilizan dentro del cuerpo de la función para realizar cálculos o ejecutar acciones.

Existen dos tipos principales de argumentos en Python:

Argumentos posicionales: Son los argumentos que se pasan a una función en el orden en que aparecen en la definición de la función. El valor del primer argumento se asigna al primer parámetro de la función, el segundo valor se asigna al segundo parámetro, y así sucesivamente.

Argumentos de palabra clave (keyword arguments): Son los argumentos que se pasan a una función con una etiqueta que indica a qué parámetro de la función se debe asignar cada valor. Esto permite pasar los argumentos en un orden diferente al de la definición de la función o incluso omitir algunos argumentos, siempre y cuando se especifiquen sus nombres.

En Python, un argumento se refiere a un valor que se pasa a una función o método cuando se llama. Los argumentos son utilizados para proporcionar datos a la función para que pueda realizar su tarea. En una definición de función, los argumentos son los valores que se colocan entre paréntesis y se utilizan dentro del cuerpo de la función para realizar cálculos o ejecutar acciones

Existen dos tipos principales de argumentos en Python:

Argumentos posicionales: Son los argumentos que se pasan a una

función en el orden en que aparecen en la definición de la función. El valor del primer argumento se asigna al primer parámetro de la función, el segundo valor se asigna al segundo parámetro, y así sucesivamente.

Argumentos de palabra clave (keyword arguments): Son los argumentos que se pasan a una función con una etiqueta que indica a qué parámetro de la función se debe asignar cada valor. Esto permite pasar los argumentos en un orden diferente al de la definición de la función o incluso omitir algunos argumentos, siempre y cuando se especifiquen sus nombres.

Aquí tienes un ejemplo de cómo se utilizan argumentos en Python:

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

# Argumentos posicionales
saludar("Juan", "Hola")

# Argumentos de palabra clave
saludar(saludo="Bonjour", nombre="Alice")
```

En este ejemplo, la función saludar toma dos argumentos: nombre y saludo. Cuando se llama a la función, se le pasan los valores "Juan" y "Hola" como argumentos posicionales en el primer caso, y los valores "Alice" y "Bonjour" como argumentos de palabra clave en el segundo caso.

Funcion LAMBDA

Las funciones Lambda son funciones anónimas que solo pueden contener una expresión.

Podrías pensar que las funciones lambda son un tema intermedio o avanzado, pero aquí aprenderás de manera sencilla como puedes comenzar a utilizarlas en tu código.

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.

Conclusión

Las funciones lambda son una manera compacta de escribir una función si solo necesitas una expresión corta. Talvez no sean algo que un programador principiante usaría, pero después de leer este artículo talvez puedas implementarlas en tu propio código.

¿Qué es un paquete pip?

pip es el sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python. Los paquetes de software son conjuntos de módulos y bibliotecas que pueden ser reutilizados en diferentes proyectos. Los paquetes pueden proporcionar funcionalidades que van desde operaciones matemáticas hasta acceso a bases de datos y análisis de datos, entre otros.

La ventaja de usar un gestor de paquetes como pip es que simplifica el proceso de instalación y actualización de paquetes, así como la gestión de dependencias entre ellos. Además, pip permite a los desarrolladores compartir sus propios paquetes con la comunidad y descargar paquetes creados por otros desarrolladores para usarlos en sus proyectos.

Características

Algunas características clave de pip incluyen:

Instalación de paquetes:

pip facilita la instalación de
paquetes de software. Puedes buscar paquetes en el Índice de Paquetes de Python (PyPI) y luego instalarlos con un simple comando.

Actualización de paquetes:

Con pip, puedes actualizar fácilmente los paquetes a la última versión disponible.

Gestión de dependencias:

pip automáticamente resuelve e instala las dependencias que un paquete necesita para funcionar.

Desinstalación de paquetes:

pip también permite desinstalar paquetes que ya no necesitas.

Listado de paquetes instalados:

Puedes usar pip para obtener una lista de todos los paquetes que tienes instalados en tu sistema.

Instalación de paquetes desde diferentes fuentes:

Además de PyPI, pip te permite instalar paquetes desde otras fuentes, como repositorios Git o archivos locales.

Ejemplos

A continuación se muestran algunos ejemplos de cómo puedes usar pip:

Instalar un paquete: pip install nombre_del_paquete

Actualizar un paquete: pip install –upgrade nombre_del_paquete

Desinstalar un paquete: pip uninstall nombre_del_paquete

Listar paquetes instalados: pip list

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.

Usos principales

Creación y distribución de paquetes propios: Además de ser una herramienta para la instalación y administración de paquetes, pip también permite a los desarrolladores crear y distribuir sus propios paquetes. Esto permite a la comunidad de Python compartir sus proyectos y código con otros.

Instalación de paquetes en un entorno virtual:

pip se puede usar junto con herramientas de entorno virtual, como virtualenv o venv, para instalar paquetes en un entorno

aislado, lo que es útil para evitar conflictos de versiones entre paquetes o para mantener separados los paquetes de diferentes proyectos.

Especificación de versiones de paquetes:

Al instalar o actualizar paquetes, puedes especificar una versión particular del paquete que deseas instalar utilizando el operador ==. Por ejemplo: pip install nombre_del_paquete==1.0.0.

Instalación de paquetes desde un archivo de requerimientos:

Puedes crear un archivo de requerimientos (usualmente llamado requirements.txt) que liste todos los paquetes y sus versiones que tu proyecto necesita. Luego, puedes usar pip para instalar todos esos paquetes de una sola vez: pip install -r requirements.txt.

Mostrar información sobre un paquete:

Puedes usar el comando pip show para mostrar información detallada sobre un paquete instalado, como su versión, ubicación, dependencias y más.

Buscar paquetes:

pip tiene un comando de búsqueda que permite buscar paquetes en el Índice de Paquetes de Python (PyPI) directamente desde la línea de comandos: pip search nombre_del_paquete.

En resumen

pip es una herramienta esencial para cualquier desarrollador de Python. Permite instalar, actualizar y administrar paquetes de software con facilidad, lo que a su vez facilita el desarrollo y la distribución de aplicaciones y proyectos de Python.

Además, pip hace que sea fácil compartir código y colaborar con otros desarrolladores en la comunidad de Python.