¿Qué es un condicional?

Un condicional es una estructura de control de flujo en programación que permite ejecutar diferentes bloques de código en función de si una condición es verdadera o falsa. En otras palabras, los condicionales permiten que un programa tome decisiones y ejecute ciertas partes de código basándose en esas decisiones.

Por ejemplo, imagina que estás desarrollando un juego de adivinanzas. El programa debe verificar si el usuario ha adivinado correctamente un número secreto. Para ello, necesitas utilizar un condicional que compare la respuesta del usuario con el número secreto. Si la respuesta es correcta, el programa debería mostrar un mensaje de felicitación; si no, debería mostrar un mensaje indicando que la respuesta es incorrecta.

Otro ejemplo más común al que se enfrenta prácticamente todo el mundo es la autentificación de usuario al entrar en cualquier plataforma. Los condicionales en este caso hacen comprobaciones sobre el nombre y contraseña de dicho usuario, lo comparan con los datos almacenados en la base de datos y, dependiendo de si la comparación es correcta o no, se ejecuta un bloque de código u otro.

En Python, los condicionales se implementan utilizando la instrucción if seguida de una expresión booleana y un bloque de código indentado. La expresión booleana se evalúa como True o False, y el bloque de código indentado solo se ejecuta si la expresión es True.

```
if condicion:
    print(‘Esto es un condicional’)
```

Python admite varios operadores lógicos para realizar las comprobaciones:

Igual: a == b
No igual: a != b
Menor: a < b
Menor o igual: a <= b
Mayor: a > b
Mayor o igual: a >= b

Y para operaciones más complejas también existen los operadores and, or y not.

and: comprueba que las condiciones a izquierda y derecha sean iguales, por ejemplo si quisiéramos acotar el rango del chequeo entre 18 y 65 podemos usar and de la siguiente forma:

```
if edad >= 18 and edad <= 65:
    print(‘Estás en edad laboral’)
```

or: a diferencia de and, donde ambas condiciones deben cumplirse, con or sólo una de ellas debe evaluar como verdadera para resultar como exitosa:

```
if edad < 18 or edad > 65:
    print(‘No estás en edad laboral’)
```

not: se usa para invertir el resultado, osea que si la comprobación es verdadera, al usar el operador devolvería como resultado False

```
if not edad >= 18:
    print(‘No tienes la mayoría de edad aún’)
```

Aparte de if, para controlar mejor el flujo del código existen otros condicionales, elif y else. Esto es útil cuando se desean hacer varias comprobaciones de forma sucesiva con la intención de que una vez se cumpla una de ellas se finalice el proceso.

Por ejemplo, volviendo a la comprobación de edad, se podría hacer de la siguiente forma usando elif y else:

```
if edad < 18:
   print(‘Todavía eres menor de edad’)
elif edad >= 18 and edad <= 65:
    print(‘Estás en edad laboral’)
else:
    print(‘Estás en edad de jubilación’)
```

Aquí se hacen 3 chequeos y se ejecutan uno detrás del otro si el anterior falla, en el mismo instante en el que uno evalúe como correcto, se cierra la comprobación. Esto quiere decir que si la edad es 17, el elif y el else se ignorarían, ahorrando así en recursos computacionales.

elif y else sólo pueden utilizarse una vez exista un if en primer lugar, tanto en el if como elif deben escribirse la condición a ser evaluada, pero con else no hace falta, se comprobará el restante del if o elif al que está adjunto, en el ejemplo anterior sería toda edad más allá de 65.

Si no hubiésemos usado elif ni else, se habrían realizado los 3 chequeos incluso si la edad fuera menor de 18.

Aparte de todo esto, los condicionales pueden anidarse cuando sea necesario simplemente añadiendo indentación. Por ejemplo si quisieramos comprobar la contraseña de usuario exclusivamente cuando el nombre es correcto se puede anidar el bloque de código dentro de esta forma:

```
if usuario_formulario == usuario_base_datos:
    # Aquí iría el código para comprobar el password
    if password_formulario == password_base_datos:
        print(‘Acceso autorizado’)
    else:
        print(‘Acceso no autorizado’)
```

Los condicionales son fundamentales a la hora de controlar el flujo de un programa y usando todo lo explicado puede especificarse de forma precisa cómo se desea que funcione.

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

Los bucles son estructuras de control que permiten ejecutar un bloque de código repetidamente mientras se cumple una condición. En Python, existen principalmente dos tipos de bucles: for y while. Estos bucles son útiles para automatizar tareas repetitivas y manipular colecciones de datos.

El bucle while se utiliza para ejecutar un bloque de código de manera repetitiva mientras se cumpla una condición específica. Su sintaxis es la siguiente:

```
while condicion:
    print(‘Dentro del loop while’)
```

Mientras se cumpla la condición el código dentro de while seguirá ejecutándose de forma ilimitada. Si no existiera una forma de salir, el programa se quedaría atascado en lo que se denomina bucle infinito, el resto del código no se ejecutaría nunca y el programa terminaría por fallar llegando en algunos casos incluso a congelar el ordenador por maximización de los recursos.

Para salir de un bucle se utiliza break, y funciona tanto para while como for.

Teniendo esto en cuenta se puede usar while para crear un juego de adivinanzas en el que hay que acertar cuál es el número secreto, un usuario introduciría un número y hasta que no acierte, el bucle se repitiría una y otra vez hasta el momento en que se halle la respuesta, momento en el que, usando break, se puede cerrar el loop.

```
while True:
    print(‘Introduce un numero de 1 a 100’)
    numero = input(‘’)
    if numero == 55:
        print(‘¡Acertaste!’)
        break
    else:
        print(‘Ése no es el número, inténtalo de nuevo’)
```

Usar break en combinación con while funciona bien en ciertas circunstancias pero por lo general el bucle se usa con o lo que se denomina una variable de control, que es una variable defenida inmediatamente antes del comienzo del loop.

Por ejemplo si queremos imprimir los numeros del 0 al 9 se escribiría de la siguiente forma:

```
numero = 0
while numero < 10:
    print(numero)
    numero += 1
```

Aquí lo que hacemos es, usando la variable de control numero, imprimir en pantalla su valor de forma repetida e incremental hasta que numero == 10, que es cuando el bucle termina al no cumplir la condición.

En este caso, es importante volver a señalar que si no se incrementase el valor de numero en 1 al final de cada pasada, el bucle se repetiría sin fin.

Combinado con break, se podría hacer que, en el momento en el que numero coincida con el valor deseado, se cierre el bucle.

```
numero = 0
while numero < 10:
    if numero == 5:
        break
    else:
        print(numero)
        numero += 1
```

En este ejemplo se imprimirían los números del 0 al 4.

Aparte de break, existe otra expresión para controlar los bucles llamada continue que en el momento en el que se usa, el bucle salta a la siguiente iteración.

Si ahora deseásemos imprimir sólo los números pares, se puede usar continue para que cada vez que numero sea par haya un salto y no se imprima en pantalla.

```
numero = 0
while numero < 10:
    if numero % 2 == 0:
        continue
    else:
        print(numero)
        numero += 1
```

Y al igual que break, continue está disponible para su uso tanto para bucles while como for.

El bucle for también se utiliza para realizar operaciones de forma repetida, pero en su caso se itera sobre una secuencia, ya sea un rango de números, una lista, una cadena de texto, una tupla o un diccionario. Por lo tanto no necesita un número de control y no es posible, o por lo menos es muy difícil, encontrarnos en la situación de crear un bucle infinito.

Usando el ejemplo anterior, imprimir los números del 0 al 9 se hace de la siguiente forma:

```
for numero in range(10):
    print(numero)
```

Por defecto a numero se le asigna el valor de 0, así que no necesitamos definirlo de forma explícita como con while, e irá incrementando su valor automáticamente con cada repetición sin nuestra intervención. Una vez numero alcance el valor de 10, el bucle se cerrará.

Al igual que antes, pueden usarse break y continue para lograr los efectos deseados.

```
for numero in range(10):
    if numero == 5:
        break
    else:
        print(numero)

for numero in range(10):
    if numero % 2 == 0:
        continue
    else:
        print(numero)
```

Ahora vamos a recorrer una lista para imprimir sus valores:

```
lista_compra = [‘pollo’, ‘leche’, ‘pan’, ‘azucar’]

for producto in lista_compra:
    print(producto)
```

En este caso lo que ocurre es que producto es por así decirlo un comodín usado en el contexto exclusivo del bucle. Se empieza a contar desde 0 y se asigna a producto el valor del índice 0 dentro de la lista. Con cada iteración se aumenta la cuenta y se reasigna a producto el nuevo valor dentro de la lista. De esta forma, por la propia funcionalidad de for, se recorre una secuencia de principio a fin sin que tengamos que introducir instrucciones adicionales, pero por si no queda lo suficientemente claro, el equivalente con while se haría de la siguiente forma:

```
contador = 0
while contador < len(lista_compra):
   print(lista_compra[contador])
   contador += 1
```

El mismo for se podría usar de forma alternativa para recorrer la lista centrándose en el número del índice.

```
for i in range(len(lista_compra)):
   print(lista_compra[i])
```

Como se puede ver los bucles son muy flexibles y permiten usar sus diversas variantes dependiendo de las circunstancias.

Esta misma flexibilidad nos permite recorrer diccionarios y utilizar tanto las claves como los valores para construir el programa tal y como deseamos.

```
saludos = {
   ‘espanol’ = ‘hola’,
   ‘italiano’ = ‘ciao’,
   ‘ingles’ = ‘hello’
}

for lengua, saludo in saludos.items():
   print(f”En {lengua} se dice {saludo}.”)
```

Al usar el método items, python es lo suficientemente inteligente para saber que queremos utilizar tanto las claves como sus valores y los asignará respectivamente en cada pasada a lengua y saludo.

Y al igual que los condicionales, los bucles pueden anidarse para cuando deseemos recorrer estructuras más complejas como una matriz.

```
matriz = [
   [0, 1, 2],
   [3, 4, 5],
   [6, 7, 8]
]
```

Se puede combinar todo lo aprendido usando while y for al mismo tiempo e imprimir todos los números de la matriz.

```
contador = 0
while contador < len(matriz):
   for numero in matriz[contador]
      print(numero)
```

¿Qué es una lista por comprensión en Python?

Una lista por comprensión (list comprehension) es una manera concisa y eficiente de crear listas. Utiliza una sola línea de código para transformar una lista existente o crear una nueva basada en una secuencia de elementos, simplificando ampliamente el código en comparación con los bucles for tradicionales.

La estructura básica de una lista por comprensión se compone de los siguientes elementos:

- Expresión: Define el elemento que se agregará a la nueva lista. Puede ser una simple variable, una operación matemática o una función.
- Bucle for: Recorre el iterable que sirve como base para la creación de la nueva lista.
- Cláusula if (opcional): Permite aplicar un filtro a los elementos del iterable, incluyendo solo aquellos que cumplen con la condición especificada.

Y se escribe de la siguiente manera

[expresion for elemento in iterable if condicion]

Ahora lo aplicamos para crear una lista de números que son el doble que los de otra lista que usaremos como base

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]

duplicados = [i * 2 for i in numeros]
```

Con esta simple línea de código, Python va a recorrer la lista numeros igualmente que si usásemos for i in numeros aparte, y a cada elemento le aplicará i * 2.

De otra forma tendríamos que escribir

```
duplicados = []

for i in numeros:
    duplicados.append(i * 2)
```

Si quisiéramos una lista con números impares necesitaríamos usar un condicional, por comprensión puede usarse la condición opcional

```
impares = [i for i in numeros if i % 2 != 0]
```

Con la estructura más básica se puede copiar otra lista

```
copia = [i for i in numeros]
```

Usando el método range pueden crearse listas de números ordenados

```
mas_numeros = [i for i in range(10)]
```

en este caso es una lista del 0 al 9 igual que numeros pero sin escribir todos los valores a mano.

Y aunque los ejemplos mostrados sean con números, también es posible partir de listas con cadenas

de texto como por ejemplo

frutas = [“manzana”, “pera”, “naranja”, “limon”, “sandia”, “melon”]

Supongamos que ahora lo que queremos es tener todos los elementos en mayúsculas, con las listas por comprensión es increíblemente sencillo:

frutas = [fruta.upper() for fruta in frutas]

Todos los valores transformados en cadenas vacías

frutas = [“” for fruta in frutas]

Todos los valores transformados en cadenas vacías excepto limon

frutas = [fruta if fruta == “limon” else “” for fruta in frutas]

O se puede usar para crear una lista con cadenas vacías (o lo que se prefiera tener como texto por defecto) con una longitud predeterminada

lista_vacia = [“” for i in range(10)]

En definitiva, las listas por comprensión son una forma concisa y eficiente de crear o modificar listas en Python.

¿Qué es un argumento en Python?

Un argumento en Python es un valor que se pasa a una función cuando se llama. Los argumentos permiten que las funciones reciban información externa, la procesen y luego devuelvan un resultado basado en esa información.

Los argumentos se escriben dentro del paréntesis al definir la función y se les proporciona un nombre tal y como haríamos al crear una variable, sirven de puente entre el interior de la función y el exterior y en algunos casos, pueden servir como variables en sí mismas.

```
def suma(argumento1, argumento2):
    return argumento1 + argumento 2

resultado = suma(1, 2)

def saludo(nombre, apellido):
    print(f”Hola {nombre} {apellido})

saludo(‘Antonio’, ‘Pérez’)
```

A no ser que se diga lo contrario, si una función tiene argumentos, es necesario llamarla con el mismo número y en el orden exacto para que funcione correctamente o el programa daría error y se cerraría.

Si se espera que una función reciba un número desconocido de argumentos se puede añadir un * delante y Python creara una lista que después podría manipularse dentro de la función.

```
def multiples_argumentos(*numeros):
    for i in numero:
        print(i)

multiples_argumentos(1, 2, 3, 4)
```

Otra alternativa es utilizar valores por defecto en el mismo argumento, por lo que si se llamase la función vacía seguiría funcionando sin problemas. Ésta es la situación que he mencionado al principio en la que el propio argumento simula ser una variable.

```
def saludo_usuario(saludo = ‘Hello’, usuario = ‘Guest’):
     print(f”{saludo} {usuario}”)

saludo_usuario(‘Hola’, ‘Fernando1999’)
saludo_usuario()
```

Ambos ejemplos se ejecutarían.

Si la lista no es suficiente, usando ** delante del argumento se pueden pasar datos en formato diccionario en la misma función sin crear el diccionario previamente.

```
def nuevo_usuario(**datos):
    for k, v in datos.items();
        print(f”{k}: {v})

nuevo_usuario(
    nombre = “Felipe”,
    edad = 55,
    ciudad = “Albacete”
)
```

Al llamar a una función es imprescindible no sólo introducir el mismo número de parámetros sino el orden también.

No obstante, existe una forma de seguir usando una función y que proporcione el resultado esperado incluso con distinto orden si a la hora de escribir el valor del argumento añadimos al inicio su nombre.

```
def login(usuario, is_admin):
    if is_admin:
        print(f“Hola {usuario}, sólo tienes permisos de lectura.”)
    else:
        print(f“Hola {usuario}, tienes permisos de administración.”)

login(is_admin = True, usuario = “PepitoPro”)
```

en vez de

```
login(“Manolo”, False)
```

Estos argumentos se llaman “keyword arguments”, los anteriores, en los que sólo se especifica el valor se llaman “positional arguments”, y en Python existe la forma de que una función acepte unos

u otros de forma específica añadiendo “*, “ o “, /”. Por lo que si queremos que el código sea más o menos estricto, no hay más que añadirlos.

Por defecto Python admite ambos indistintamente:

```
def print_name(nombre):
    print(f“Hola {nombre}”)

print_name(“Rober”)
```

funcionaría al igual que

```
print_name(nombre = “Rober”)
```

Ahora si quisiéramos usar sólo keyword argument, la función se escribiría

```
def print_name(*, nombre):
    print(f“Hola {nombre}”)
```

Por lo que sólo se podría usar print_name(nombre = “Rober”)

Si por el contrario se desea usar positional arguments sería

```
def print_name(nombre, /):
    print(f“Hola {nombre}”)

print_name(“Rober”)
```

Al usar positional arguments, por consiguiente, se perdería la flexibilidad de poder introducir los argumentos en orden aleatorio.

¿Qué es una función Lambda en Python?

Las funciones lambda, también conocidas como funciones anónimas, son una forma compacta de definir funciones pequeñas y sencillas. A diferencia de las funciones tradicionales que se definen con la palabra clave def, las funciones lambda no tienen nombre y se escriben en una sola línea.

Su sintaxis es entonces

```
lambda argumentos: expresion
```

argumentos representa los parámetros que la función lambda puede recibir. Se separan por comas y no es obligatorio incluirlos si la función no requiere argumentos.

expresion define la operación que se va a realizar con los argumentos. Se puede componer de operadores aritméticos, lógicos, comparaciones, llamadas a otras funciones, etc.

Al ser anónimas, por norma general se almacenan en variables ya que no se pueden llamar de la misma forma que con las funciones normales.

```
suma = lambda numero1, numero2: numero1 + numero2

print(suma(1, 2))
```

Son ideales para operaciones que se pueden describir en una sola expresión o para utilizarlas como argumento en otros métodos como filter o map para manipular listas.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si quisiéramos crear un lista usando como base numeros y duplicando todos sus valores, una función lambda sería muy útil en combinación con map

```
duplicados = list(map(lambda i: i * 2, numeros))
```

Y si sólo quisiera los valores pares, se puede combinar con filter

```
pares = list(filter(lambda i: i % 2 == 0, numeros))
```

¿Qué es un paquete pip?

En el mundo de Python, pip, que es un acrónimo de Pip Installs Packages, es una herramienta para la instalación y gestión de librerías y paquetes de software. Se trata de un sistema de gestión de paquetes que facilita la búsqueda, descarga e instalación de módulos externos que amplían las funcionalidades del lenguaje Python.

Se pueden encontrar una gran variedad de paquetes en el índice de paquetes de Python (PyPI), un repositorio online que alberga módulos para diversas tareas, desde análisis de datos y visualización hasta desarrollo web y machine learning.

Pip se utiliza para simplificar el proceso de instalación de paquetes que no forman parte de la biblioteca estándar de Python. Esto es especialmente útil para desarrolladores que necesitan utilizar bibliotecas específicas para sus proyectos sin tener que preocuparse por las dependencias manualmente.

Un paquete pip es por lo tanto, código escrito en Python por otras personas para añadir funcionalidad adicional y que podemos descargar e instalar con comandos pip.

Por ejemplo, si en nuestra máquina no existiera pandas, una librería para análisis de datos, al usar import pandas dentro de nuestro código, nos daría un error.

Deberíamos usar pip para instalar el paquete previamente mediantes el comandos

```
pip install pandas
```

Otros comandos comunes son:

```
pip install –upgrade pandas     # para actualizar
pip list                        # para listar los paquetes
pip uninstall pandas            # para desinstalar
pip search pandas               # para buscar si existe el paquete pandas
pip show pandas                 # para mostrar información
```