Condicionales en Python:

Una condicional es una estructura de control que determina qué acción debe ejecutar el código dependiendo de si las condiciones establecidas son verdaderas (True) o falsas (False).

La sentencia condicional en Python (al igual que en muchos otros lenguajes de programación) es *if*. La sintaxis *if* seguida de la condición seguida de dos puntos:

Ejemplo 1

```
edad = 13 # variable que queremos comprobar

if edad < 16: # condición
    print("Eres menor y no puedes entrar") # acción si la condición es cierta
```

En el ejemplo anterior, si la condición se cumple (*edad <16* es *True*) se mostrará el texto "Eres menor y no puedes trabajar" pero si la condición no se cumple (*False*) el código no indica que se realice ninguna acción. Si queremos añadir una acción para cualquier otro caso que no se corresponda con la condición debemos añadir *else:*

Ejemplo 2

```
edad = 13 # variable que queremos comprobar

if edad < 16: # condición
        print("Eres menor y no puedes entrar") # acción si la condición es cierta
else:
        print("¡Bienvenido!") # acción para el resto de los casos
```

Es posible que también se quiera añadir diferentes comprobaciones entre la primera condicional (*if*) y la última (*else*). En este caso se añade la condición (o condiciones) precedida de la palabra clave *elif* (combinación de *else if*).

Ejemplo 3

```
edad = 13 # variable que queremos comprobar

if edad < 18: # condición
        print("Eres menor y no puedes entrar.") # acción si la condición es cierta
elif edad >= 18 and edad < =30:  # segunda condición
    print("Si eres estudiante puedes beneficiarte del descuento.") # acción
elif edad >= 65:  # tercera condición
    print("Si estás jubilado puedes beneficiarte del descuento.") # acción
else:
        print(¡Bienvenido! ") # acción para el resto de los casos
```

El código anterior se ejecutará de manera ordenada, es decir, primero se comprueba la condición que comienza con *if.* Si es cierta (*True*) se ejecutará la acción correspondiente. Si, por el contrario, no es cierta (*False*) se comprobará la condición que sigue al primer *elif*. Si esta es cierta, se ejecutará la acción asociada. Si no lo es se pasará a comprobar la siguiente condición, y así hasta llegar al final del código.

Operadores de comparación

Cuando escribimos condiciones debemos incluir alguna expresión de comparación usando los siguientes operadores:

Operador	Símbolo	Uso con
Igualdad	==	Cadenas (*string*) y números
Desigualdad	!=	Cadenas (*string*) y números
Menor que	<	Números
Menor o igual que	<=	Números
Mayor que	>	Números
Mayor o igual que	>=	Números

Condicionales anidados

Python permite establecer una condicional en el interior de otra. A eso se le llama condiciones anidados, Se pueden anidar cuantos condicionales se necesita, aunque se recomienda no incluir más de dos o tres.

El ejemplo anterior comprueba varias condiciones por lo que también podría haber sido expresada como una condición anidada:

Ejemplo 4

```
edad = 13

if edad < 18:
        print("Eres menor y no puedes entrar.")
else:
        if  edad >= 18 and edad  <=30:
                print("Si eres estudiante puedes beneficiarte del descuento.")
        else:
                 if edad >= 65:  # cuarta condición
                        print("Si estás jubilado puedes beneficiarte del descuento.")
                else:
                        print("¡Bienvenido!") # acción para el resto de los casos
```

Los condicionales anidados nos permiten reaccionar a cada caso, pues mostramos un mensaje acorde a cada condición que se cumple. Sin embargo, como podemos observar, el código en este ejemplo es menos intuitivo. El uso de *elif* ayuda a simplificar la sintaxis y la compresión del código.

Condicionales compuestos

Los condicionales compuestos consisten en unir condiciones dentro de una clave *if* mediante las palabras clave *and* (y) u *or* (o). En el primer caso todas las condiciones deben ser cierta (*True*). En el segundo sólo una de las condiciones debe cumplirse.

En el ejemplo anterior, `elif edad >= 18 and edad < =30:` es un caso de condicional compuesto.

Asignación condicional de una única línea (operador ternario)

El operador ternario es una declaración *if-else* que se define en una sola línea. Una condicional simple en la que sólo se establece una condición y las acciones correspondientes al resultado de *True* o *False* pueden adoptar la sintaxis de una única línea. Por ejemplo, tomando como ejemplo un caso anterior:

```
edad = 13

if edad < 16:
        mensaje = "Eres menor y no puedes entrar"
else:
        mensaje = "¡Bienvenido!"

print(mensaje)
```

Puede adoptar la siguiente sintaxis: [acción si *True*] *if* [condición] *else* [acción si *False*]

Ejemplo 5

```
edad = 13

mensaje = "Eres menor y no puedes entrar" if edad < 16 else "¡Bienvenido!"

print(mensaje)
```

El resultado es un código más conciso sin reducir la comprensión.

Bucles en Python:

Los bucles permiten que varios elementos combinados en una misma "colección" sean procesados individualmente según un mismo esquema. Nos permiten ejecutar un bloque de código varias veces seguidas de forma automática. Esto se traduce en un código más eficiente, compacto y flexible.

En Python existen dos tipos principales de bucles: *for* y *while*.

Bucle *for*

El bucle *for* se utiliza para recorrer los elementos de un objeto iterable (lista, tupla, conjunto, diccionario, ...) y ejecutar un bloque de código, lo que se denomina "iteración". En cada paso de la iteración se tiene en cuenta a un único elemento del objeto iterable, sobre el cuál se pueden aplicar una serie de operaciones. Un *iterable* es un objeto que permite recorrer sus elementos uno a uno. Un *iterador* es un objeto que define un mecanismo para recorrer los elementos del iterable asociado.

La sintaxis básica del bucle *for* es la siguiente:

```
for <elemento> in <objeto iterable>:
    <código>
```

El bucle finaliza su ejecución cuando se recorren todos los elementos del objeto iterable. Por ejemplo:

Ejemplo 1

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

for num in numeros: # para (for) cada elemento (num) en el objeto iterable (la lista numeros)
    print(num) #acción a realizar. Se imprimirá cada número en la lista.
```

También podemos incluir una lógica más compleja en el cuerpo de un bucle *for*

Ejemplo 2

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

for num in numeros: # para (for) cada elemento (num) en el objeto iterable (la lista numeros)
    calc = num**2 - (num-1)*(num+1) # cálculo
    print(calc) #acción
```

Bucle *for* en diccionarios

Un diccionario está compuesto por pares clave/valor, por lo que hay distintas formas de iterar sobre ellas.

1 – Recorrer las claves del diccionario.

Ejemplo 3

```
capitales_de_paises = {"España": "Madrid", "Francia": "Paris", "Italia": "Roma"}

for pais in capitales_de_paises:
    print(pais)
```

2 - Iterar sobre los valores del diccionario.

Ejemplo 4

```
capitales_de_paises = {"España": "Madrid", "Francia": "Paris", "Italia": "Roma"}

for capital in capitales_de_paises.values():
        print(capital)
```

3 – Iterar a la vez sobre la clave y el valor de cada uno de los elementos del diccionario.

Ejemplo 5

```
capitales_de_paises = {"España": "Madrid", "Francia": "Paris", "Italia": "Roma"}

for pais, capital in capitales_de_paises.items():
        print(pais+":", capital)
```

Python *for* y la clase range

Para los casos en los que se quiera realizar un bucle *for* en una secuencia numérica, se puede utilizar la clase *range*.

El tipo de datos *range* se puede invocar con uno, dos e incluso tres parámetros:

- range (max): Un iterable de números enteros consecutivos que empieza en *0* y acaba en el valor máximo menos uno (*max-1*)

Ejemplo 6

```
for num in range(10):
        print (num) # imprimirá los números desde el 0 al 9.
```

- range(min, max): Un iterable de números enteros consecutivos que empieza en el valor mínimo (*min*) y acaba en el valor máximo menos uno (*max-1*)

Ejemplo 7

```
for num in range(4,10):
        print (num) # imprimirá los números desde el 4 al 9.
```

- range(min, max, step): Un iterable de números enteros consecutivos que empieza el valor mínimo (*min*), acaba en el valor máximo menos uno (*max-1*) y los valores se van incrementando paso a paso (*step by step)*. Este último caso simula el bucle *for* con variable de control.

Ejemplo 8

```
for num in range(1,10, 2):
        print (num) # imprimirá los números del 1 al 9, cada dos números (es decir, sólo los números impares)
```

Bucle *for* con *break* y *continue*

La iteración del bucle *for* se puede modificar mediante el uso de las sentencias *break* y *continue*.

- *break* se utiliza para finalizar y salir del bucle si se cumple la condición.

Ejemplo 9

```
nombre = "Enrique"
lista_nombre = ["Jessica", "Paul", "George", "Henry", "Adán"]

for item in lista_nombre:
    if item == nombre: # condición
        print(f"{nombre} está en la lista.") # acción si condición es True
        break # si condición es True el bucle se termina y se sale de él.
    else:
        print(f"Buenos días {item}.") # acción si condición es False
```

- *continue* se utiliza para saltar al siguiente paso de la iteración, ignorando todas las sentencias que le siguen y que forman parte del bucle.

Ejemplo 10

```
nombre = "Enrique"
lista_nombre = ["Jessica", "Paul", "George", "Henry", "Adán"]

for item in lista_nombre:
    if item == nombre: # condición
        print(f"{nombre} está en la lista.") # acción si condición es True
        continue # el bucle pasa al siguiente elemento.
    else:
        print(f"Buenos días {item}.") # acción si condición es False
```

Bucle *while*

El bucle *while* se utiliza para ejecutar un bloque de código tantas veces como la condición específica sea verdadera. Utilizamos un bucle *while* cuando el tamaño de la colección no puede determinarse en el momento de ejecutar el programa.

La estructura de esta sentencia *while* es la siguiente:

```
while <condición>:
    <código>
```

Ejemplo 11

```
numeros = [1, 2, 3, 4, 5]
contador = 0

while contador < 5: # la condición que se evalúa en cada iteración del bucle. Mientras la condición sea True, la acción o acciones se realizarán.
    print(contador) # acción
    contador += 1  # acción
```

Este bucle *while* imprimirá los números del 0 al 4, ya que incrementamos el contador en cada iteración. El bucle continuará ejecutándose mientras la condición (contador < 5) sea verdadera (*True*). Una vez que la condición se vuelva falsa (False) el bucle se detendrá.

Bucle while con *break* / *continue* y *else*

El bucle *while* también puede ser alterado con las sentencias *break* y *continue*.

Ejemplo 12

```
lista_nombre = ["Jessica", "Paul", "George", "Henry", "Adam"]

contador = 0

while contador < len(lista_nombre): # primera condición
        nombre = lista_nombre[contador]
        if nombre == "Enrique":  # segunda condición
                print(f'{nombre} ha sido encontrado en la posición número {contador+1}')
                break # el bucle se termina si esta segunda condición es True
         else: # en caso de que esta segunda condición sea False
                contador += 1
else: # en caso de que la primera condición sea False
        print(f"Enrique no se encuentra en la lista.")
```

Por otro lado, como el ejemplo muestra, al bucle *while* le podemos añadir la sentencia opcional *else*. El bloque de código bajo esta sentencia se ejecutará siempre y cuando la condición de la sentencia *while* sea *False* y no se haya ejecutado una sentencia *break*.

Lista por comprensión:

La lista por comprensión ayuda a crear una nueva lista a partir de objetos iterables existentes (listas, cadenas y tuplas) aplicando una expresión/condición a cada elemento para determinar qué elementos se incluirán en la lista resultante

La sintaxis (para una lista) es la siguiente:

```
<new_list> = [<expression> for <item> in <iterable>]
```

Esta sintaxis permite un código más legible, eficiente y conciso que los bucles *for* en modelos sencillos, es decir, donde la expresión no es compleja. Por ejemplo, tenemos una lista y queremos crear una nueva donde los elementos sean el doble de los elementos iniciales.

Usando el bucle *for …. in* lo haríamos así:

Ejemplo 1

```
nums = [1, 2, 3, 4, 5]
numeros_dobles = []

for num in nums:
    numeros_dobles.append(num*2)

print(numeros_dobles) #Output: [1, 4, 6, 8,10]
```

Utilizando el método de lista por comprensión el código quedaría así:

Ejemplo 2

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

numeros_dobles = [num*2 for num in nums] # indicamos que cada elemento de la lista se multiplique por dos.

print(numeros_dobles) #Output: [1, 4, 6, 8,10]
```

También se puede utilizar este método con condicionales. En el siguiente ejemplo se busca extraer sólo los números impares que hay en el rango de valores del 0 al 9.

El código con el bucle *for* sería:

Ejemplo 3

```
numeros_impares = []
for num in range(10):
    if num % 2 != 0: # si el resto de la división del número entre 2 es distinto a cero entonces el número es impar
    numeros_impares.append(num)

print(numeros_impares) # Output: [1, 3, 5, 7, 9]
```

Argumento:

Un argumento es un valor que se pasa a una función o método cuando este se llama, es decir, para llevar a cabo la acción específica. Los argumentos permiten a las funciones ser flexibles y poder ser utilizadas en varias veces, ya que los valores de los argumentos pueden ser diferentes cada vez que se llama a la función.

Hay diferentes tipos de argumentos:

1. **Argumentos posicionales**: se pasan a una función en un orden específico, basado en la posición de los parámetros definidos en la función. El número de argumentos y su orden deben coincidir con los parámetros definidos en la función.

Ejemplo 1

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

resultado = suma(3, 5) # 3 se asigna a a y 5 se asigna a b
print(resultado) # Output: 8
```

2. **Argumentos de palabra clave (*keyword arguments*):** utilizan el nombre del parámetro al que se quiere asignar el valor, lo que permite cambiar el orden de los argumentos sin afectar el resultado.

Ejemplo 2

```
def saludar(nombre, saludo):
    return f"{saludo}, {nombre}:"

mensaje = saludar(saludo="Hola", nombre="Juan")
print(mensaje) # Output: "Hola, Juan:"
```

3. **Argumentos por defecto**: tienen un valor predeterminado. Si no se proporciona un valor para ese argumento al llamar a la función, se utilizará el valor predeterminado.

Ejemplo 3

```
def saludar(nombre, saludo="Hola"):
    return f"{saludo}, {nombre}."

mensaje = saludar("Sara")
print(mensaje) # Output: "Hola, Sara."
```

4. **Argumentos arbitrarios (*args y **kwargs):**
    - ***args**: Permite pasar un número variable de argumentos posicionales a una función. Los argumentos se recogen como una tupla dentro de la función.

Ejemplo 4

```
def promedio(*numeros):
    return sum(numeros) / len(numeros) # se suman los valores de los argumentos y se dividen entre el número de argumentos

resultado = promedio(5, 10, 15) # se definen los valores de los argumentos. Se especifican tantos como se desee.
print(resultado) # Output: 10.0
```

- ****kwargs**: Permite pasar un número variable de argumentos de palabra clave a una función. Los argumentos se recogen como un diccionario dentro de la función.

Ejemplo 5

```
def detalles(**info):
    for key, value in info.items(): # formato elementos de un diccionario
        print(f"{key}: {value}")

detalles(nombre="Ana", edad=30, ciudad="Madrid") # los valores para cada elemento del diccionario.

# Output:
# nombre: Ana
# edad: 30
# ciudad: Madrid.
```

Con el método de la lista por comprehensión, el código sería:

Ejemplo 4

```
numeros_impares = [num for num in range(10) if num % 2 != 0]

print(numeros_impares) # Output: [1, 3, 5, 7, 9]
```

Función *lambda*:

Una función *lambda* es una función anónima y de una sola línea que puede definirse de manera concisa utilizando la palabra clave *lambda*. A diferencia de las funciones definidas con *def*, las funciones *lambda* no requieren un nombre específico y se utilizan principalmente en situaciones donde se necesita una función rápida y temporal.

Por ejemplo: se quiere hacer una función de suma de dos parámetros x e y. El código de una función *def* sería:

Ejemplo 1

```
def suma(x, y):
        return x+y

resultado = suma(3, 5)
print(resultado) # Output: 8
```

El código correspondiente a una función *lambda*:

Ejemplo 2

```
suma = lambda x, y: x + y # la función lambda se asigna a la variable suma y luego se puede llamar como cualquier otra función.

resultado = suma(3, 5)
print(resultado) # Output: 8
```

Paquete *pip*:

pip es el sistema estándar de gestión de paquetes de software desarrollados para Python, es decir, permite instalar, actualizar y administrar esos paquetes. Un paquete es un conjunto de módulos o subpaquetes de bibliotecas y herramientas adicionales de código, datos o documentación que están agrupados y que ofrecen funcionalidades específicas. Una vez instalados pueden ser utilizados varias veces.

Para instalar un paquete utilizando *pip*, se utiliza el siguiente comando en la línea de comandos:

```
pip install nombre_del_paquete
```

pip instala el paquete solicitado y cualquier otro paquete o paquetes necesarios para que el paquete funcione correctamente. La mayoría de los paquetes se encuentran en el repositorio PyPI.