



Curso de Especialización de «Desarrollo de Aplicaciones en Lenguaje Python»

Avanza 2024/25

Módulo «Estructuras de control en Python»

Unidad de trabajo 2

Sentencias condicionales

Profesor: Ismael Reyes Rodríguez

Tabla de contenido

1	Objetivos	3
2	Concepto de sentencia condicional	4
2.1	Bloques de código	4
2.2	Operadores condicionales y lógicos.....	6
2.3	Precedencia de operadores.....	7
2.4	Otros aspectos de la programación en Python	8
2.5	Comentarios	8
2.6	Ancho de código	9
3	Sentencia condicional «if»	10
3.1	Sentencia «elif»	11
3.2	Asignaciones condicionales	12
3.3	Operador morsa.....	14
3.4	Otras características de las condiciones.....	15
3.4.1	Comparación múltiple en una línea.....	15
3.4.2	Condición sobre un rango o una lista	16
3.4.3	Cortocircuito lógico.....	16
3.4.4	Booleanos en condiciones.....	18
3.4.5	Valor nulo. «None»	19
3.4.6	Veracidad.....	21
3.4.7	Asignación lógica	22
4	Sentencia condicional «match-case»	23
4.1	Patrones con tuplas.....	24
4.2	Patrones con diccionarios	27
4.3	Otras particularidades.....	29
4.4	Ejemplos de uso	30
5	Web para explorar.....	32

1 Objetivos

El objetivo principal que se conseguirá con la realización de esta unidad de trabajo será el de reconocer las sentencias condicionales en Python aplicándolas a la resolución de problemas que impliquen toma de decisiones.

Para la consecución de dicho objetivo se trabajarán varios aspectos entre los que destacan:

- Concepto de sentencia condicional.
- Partes de una sentencia condicional.
- Sangrado.
- Ejecuciones condicional y control de variables.
- Funcionamiento de las sentencias condicionales.
- Aplicación de sentencias condicionales.
- Anidaciones.
- Sintaxis a aplicar en estructuras compactas.
- Bloques de programas en sentencias condicionales.
- Bloques de programas en sentencias condicionales anidadas.

Muchos de estos aspectos ya se han tratado en la Unidad de trabajo 1, pero se analizarán con más detenimiento en este tema.

2 Concepto de sentencia condicional

Una **sentencia condicional** es una estructura de control que permite ejecutar un bloque de código basándose en si una condición especificada es verdadera (*True*) o falsa (*False*). Estas sentencias son esenciales para tomar decisiones en la ejecución del programa.

En definitiva, una sentencia condicional permite que el flujo de un programa cambie dependiendo del resultado de una condición lógica. En Python, como vimos en el tema anterior las sentencias «*if*» y «*match-case*» son sentencias condicionales que ejecutan un bloque u otro dependiendo de una condición.

En la propia definición de sentencia condicional vemos conceptos como «*bloque*» o «*condición*» sobre los que ya hemos trabajado. El primero hace referencia a un conjunto de sentencias que se ejecutan secuencialmente y, el segundo, a partes de código que se evalúan y devuelven un valor lógico, verdadero o falso.

Los siguientes apartados profundizan sobre estos conceptos.

2.1 Bloques de código

Como se vio en el tema anterior, un **bloque** de código en Python es un conjunto de líneas de código que se agrupan y se ejecutan juntas como una unidad lógica. Los bloques de código se utilizan para definir funciones, bucles, condiciones, y otras estructuras de control.

En Python, los bloques de código se **delimitan por la indentación**, es decir, el espacio que se deja al comienzo de cada línea. A diferencia de otros lenguajes que usan llaves (`{}`) o palabras clave como *begin* y *end*, Python utiliza sangrías para indicar la jerarquía y estructura del programa.

Las características del bloque de código en Python son:

- **Indentación Obligatoria:**
 - Python utiliza indentaciones (espacios o tabulaciones) para definir bloques. Por convención, se suelen usar **4 espacios** por nivel de indentación.
 - Todas las líneas de un bloque deben estar al mismo nivel de indentación.
- **Errores por Mala Indentación:**
 - La falta de coherencia en la indentación genera errores como *IndentationError*.

- **Uso en Estructuras Comunes:**

- Los bloques son fundamentales para construir funciones, bucles (for, while), y estructuras condicionales (if, elif, else), como se mostrará a lo largo de este tema.

El siguiente ejemplo muestra dos bloques de código dentro de un bucle `for`:

```
for i in range(3):  
    print(f"Iteración {i}")  
    if i == 1:  
        print("Estamos en la segunda iteración")
```

Bloque 1: Bajo el bucle `for` (`print` y el condicional `if`).

Bloque 2: Bajo el `if` (`print("Estamos en la segunda iteración")`).

Cada nivel de indentación indica un bloque jerárquico dentro del anterior.

Los **errores** en un programa Python pueden ser debidos a una mala indentación del código. El siguiente ejemplo mostraría un “*Error: IndentationError: unexpected indent*”:

```
x = 5  
if x > 3: # Esta línea tiene un espacio extra al inicio  
    print("X es mayor que 3")
```

Esto ocurre porque el bloque del `if` no sigue la indentación estándar.

Es conveniente tener en cuenta los siguientes consejos sobre la indentación:

- Usar 4 espacios por nivel sin mezclar espacios y tabulaciones.
- Configurar el editor, ya que la mayoría de los editores (como VSCode o PyCharm) tienen herramientas para manejar la indentación correctamente.
- Antes de ejecutar un programa, se debe verificar que todos los bloques sigan la misma estructura de indentación.

En definitiva, recuerda que Python prioriza la claridad y la simplicidad en su sintaxis, y la indentación es una parte esencial para lograr un código limpio y legible.

2.2 Operadores condicionales y lógicos

A modo de recuerdo, este punto repasa qué **operadores condicionales** se utilizan en Python:

Comparación	Operador	Ejemplo	Resultado del ejemplo
Igual que	<code>==</code>	<code>3 == 3</code>	True
Diferente a	<code>!=</code>	<code>4 != 4</code>	False
Mayor que	<code>></code>	<code>5 > 1</code>	True
Mayor o igual que	<code>>=</code>	<code>3 >= 3</code>	True
Menor que	<code><</code>	<code>5 < 1</code>	False
Menor o igual que	<code><=</code>	<code>5 <= 10</code>	True

Y también los **operadores lógicos**, de la que vemos su tabla de verdad:

X	Y	X and Y	X or Y	Not X
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Sin olvidar, como vimos en el documento de “*Conceptos previos y recomendaciones.pdf*”, que también disponemos de diferentes **operadores aritméticos**:

Operación	Operador
Suma	<code>+</code>
Resta	<code>-</code>
Multiplicación	<code>*</code>
División	<code>/</code>
División entera	<code>//</code>
Módulo	<code>%</code>
Exponenciación	<code>**</code>

Todos estos operadores pueden combinarse para construir una expresión condicional, como se muestra en el siguiente ejemplo:

```
temperatura = 25
lluvia = False
dia = "lunes"

if (temperatura > 20 and not lluvia) or dia == "domingo":
    print("Día perfecto para salir.")
else:
    print("Mejor quedarse en casa.")
```

2.3 Precedencia de operadores

La **precedencia de operadores en Python** determina el orden en el que se evalúan los operadores en una expresión. Los operadores con mayor precedencia se evalúan antes que los de menor precedencia. Si los operadores tienen la misma precedencia, se evalúan de **izquierda a derecha** (excepto algunos, como la asignación, que son de derecha a izquierda).

La siguiente tabla muestra el orden de precedencia de mayor a menor:

Precedencia	Operador(es)	Descripción
1 (Mayor)	()	Paréntesis: Fuerzan el orden de evaluación.
2	**	Exponenciación.
3	+x, -x, ~x	Operadores unarios: Positivo, Negativo, Complemento.
4	*, /, //, %	Multiplicación, División, División entera, Módulo.
5	+, -	Suma, Resta.
6	<<, >>	Desplazamiento de bits: Izquierda, Derecha.
7	&	AND bit a bit.
8	^	XOR bit a bit.
9	~	~
10	Comparaciones: ==, !=, <, <=, >, >=, is, is not, in, not in	Comparaciones.
11	not	Negación lógica.
12	and	AND lógico.
13 (Menor)	or	OR lógico.

A modo de resumen:

1. **Paréntesis** tienen la precedencia más alta, y se evalúan primero.
2. **Exponenciación (**)** tiene mayor precedencia que la multiplicación o suma.
3. **Operadores unarios (+, -)** se evalúan antes de las operaciones aritméticas binarias.
4. **Comparaciones** como <, >, y == tienen menor precedencia que los operadores aritméticos.
5. **Operadores lógicos** (not, and, or) tienen las precedencias más bajas.

2.4 Otros aspectos de la programación en Python

Aunque directamente no tienen relación con sentencias condicionales, a continuación, se exponen algunas características del código Python que conviene saber.

2.5 Comentarios

Los **comentarios** son anotaciones que podemos incluir en nuestro programa y que nos permiten aclarar ciertos aspectos del código. Estas indicaciones son ignoradas por el intérprete de Python.

Los comentarios se incluyen usando el símbolo almohadilla **#** y comprenden hasta el final de la línea, como se aprecia en el siguiente ejemplo:

```
# Edad del universo en días
universe_age = 13800 * (10 ** 6) * 365
```

Es conveniente utilizar los comentarios necesarios para aclarar cualquier aspecto del código o indicar, por ejemplo, el nombre del autor y la fecha de creación de un programa. Es conveniente considerar las siguientes reglas de cara a escribir buenos comentarios:

1. Los comentarios no deberían duplicar el código.
2. Los buenos comentarios no arreglan un código poco claro.
3. Si no puedes escribir un comentario claro, puede haber un problema en el código.
4. Los comentarios deberían evitar la confusión, no crearla.
5. Usa comentarios para explicar código no idiomático.
6. Proporciona enlaces a la fuente original del código copiado.
7. Incluye enlaces a referencias externas que sean de ayuda.
8. Añade comentarios cuando arregles errores.
9. Usa comentarios para destacar implementaciones incompletas.

Se pueden crear comentarios de más de una línea añadiendo a ellas la almohadilla:

```
# Este programa calcula el área de un círculo.

# Para ello, se utiliza la fórmula:

# área = pi * radio ** 2

import math

radio = 5

area = math.pi * radio ** 2

print(f"El área del círculo es: {area}")
```

También es posible utilizar cadenas de texto delimitadas por comillas triples ("""" o """) para incluir comentarios multilínea. Aunque su uso principal es documentar funciones, clases o módulos, también se pueden usar como comentarios largos.

```
"""
```

Este programa realiza lo siguiente:

1. Calcula el área de un círculo.
2. Imprime el resultado.

```
"""
```

```
import math

radio = 5

area = math.pi * radio ** 2

print(f"El área del círculo es: {area}")
```

Nota: Este método técnicamente crea un objeto `str` en el código que no se asigna a ninguna variable, pero no tiene impacto en la ejecución del programa.

2.6 Ancho de código

Los programas suelen ser más legibles cuando las líneas no son excesivamente largas. La longitud máxima de línea recomendada por [la guía de estilo de Python](#) es de **80 caracteres**.

Sin embargo, esto genera una cierta polémica hoy en día, ya que los tamaños de pantalla han aumentado y las resoluciones son mucho mayores que hace años. Así las líneas de más de 80 caracteres se siguen visualizando correctamente. Hay personas que son más estrictas en este límite y otras más flexibles.

En caso de que queramos **romper una línea de código** demasiado larga, podemos:

1. Usar la *barra invertida* \:

```
factorial = 4 * 3 * 2 * 1
factorial = 4 * \
            3 * \
            2 * \
            1
```

2. Usar los paréntesis (...):

```
factorial = (4 * \
            3 * \
            2 * \
            1)
```

3 Sentencia condicional «if»

Comenzamos este apartado con un resumen de lo que se estudió en el tema anterior.

La sentencia condicional en Python es **if**. En su escritura debemos añadir una **expresión de comparación** terminando con dos puntos al final de la línea, como vemos en el ejemplo:

```
temperature = 40
if temperature > 35:
    print('Aviso por alta temperatura')
```

Salida: Aviso por alta temperatura

Para ejecutar un bloque cuando no se cumple la condición se utiliza la sentencia **else**.

Veamos el mismo ejemplo anterior, pero añadiendo esta variante:

```
temperature = 20
if temperature > 35:
    print('Aviso por alta temperatura')
else:
    print('Parámetros normales')
```

Salida: Parámetros normales

En el siguiente ejemplo se aprecia cómo se pueden **anidar condiciones**:

```
temperature = 28
if temperature < 20:
    if temperature < 10:
        print('Nivel azul')
    else:
        print('Nivel verde')
else:
    if temperature < 30:
        print('Nivel naranja')
    else:
        print('Nivel rojo')
```

Salida: Nivel naranja

Un aspecto a tener en cuenta es que, en Python no puede haber un bloque *if* vacío. El siguiente código daría un *SyntaxError*.

```
if a > 5:
```

Por lo tanto, si tenemos un *if* sin contenido, tal vez porque sea una tarea pendiente que estamos dejando para implementar en un futuro, es necesario hacer uso de **pass** para evitar el error. Realmente *pass* no hace nada, simplemente es para tener contenido al intérprete de código.

```
if a > 5:  
    pass
```

Con esto evitariamos el error

Algo no demasiado recomendable pero que es posible, es poner todo el bloque que va dentro del *if* en la misma línea, tras los “dos puntos” (:). Si el bloque de código no es muy largo, puede ser útil para ahorrarse alguna línea de código.

```
if a > 5: print("Es > 5")
```

Si el bloque de código tiene más de una línea, se pueden poner también en la misma línea separándolas con “punto y coma” (;).

```
if a > 5: print("Es > 5"); print("Dentro del if")
```

3.1 Sentencia «*elif*»

Python nos ofrece una mejora en la escritura de condiciones anidadas cuando aparecen consecutivamente un *else* y un *if*. Podemos sustituirlos por la sentencia ***elif***:



*Construcción de la sentencia «*elif*»*

El siguiente ejemplo muestra cómo se puede aplicar esta sentencia al código anterior:

```
temperature = 28

if temperature < 20:
    if temperature < 10:
        print('Nivel azul')
    else:
        print('Nivel verde')
elif temperature < 30:
    print('Nivel naranja')
else:
    print('Nivel rojo')
```

Salida: Nivel naranja

3.2 Asignaciones condicionales

Supongamos que queremos asignar un nivel de riesgo de incendio en función de la temperatura. En su **versión clásica** se podría escribir como:

```
temperature = 35

if temperature < 30:
    print('LOW')
else:
    print('HIGH')
```

Salida: HIGH

Sin embargo, esto lo podríamos abbreviar con una **asignación condicional de una única línea**:

```
print('LOW' if temperature < 30 else 'HIGH')
```

Salida: HIGH

De una manera formal, una **asignación condicional en una única línea** se realiza mediante la expresión **ternaria**. Este formato permite evaluar una condición y asignar un valor dependiendo del resultado de esa condición.

```
variable = valor_si_verdadero if condicion else valor_si_falso
```

A continuación, se muestra otro ejemplo de este tipo de asignaciones:

```
x = 10
resultado = "positivo" if x > 0 else "negativo"
print(resultado)
```

Algunos detalles importantes que se deben tener en cuenta son:

1. **Orden** de los elementos:

- La **condición** se coloca en el medio, separando los dos valores posibles.
- Esto difiere de lenguajes como C o Java, donde la condición va al inicio.

2. Cláusula **else obligatoria**:

- A diferencia de las estructuras *if* normales, las expresiones ternarias requieren un bloque *else*. Sin él, se producirá un error de sintaxis.

3. Para mantener **legibilidad**:

- Las asignaciones ternarias son ideales para expresiones simples. Si la condición o los valores son complejos, es mejor usar un bloque *if-else* tradicional.

3.3 Operador morsa

A partir de Python 3.8 se incorpora el [operador morsa](#) que permite unificar sentencias de asignación dentro de expresiones. Su nombre proviene de la forma que adquiere:

`:=`

Veamos un ejemplo en el que computamos el perímetro de una circunferencia, indicando al usuario que debe incrementarlo siempre y cuando no llegue a un mínimo establecido.

Utilizando la **versión tradicional**:

```
radius = 4.25
perimeter = 2 * 3.14 * radius
if perimeter < 100:
    print('Incrementa el radio para alcanzar el perímetro mínimo')
    print('Perímetro actual: ', perimeter)
```

Salida: Incrementa el radio para alcanzar el perímetro mínimo

Perímetro actual: 26.69

Utilizando el **operador morsa**:

```
radius = 4.25
if (perimeter := 2 * 3.14 * radius) < 100:
    print('Incrementa el radio para alcanzar el perímetro mínimo')
    print('Perímetro actual: ', perimeter)
```

Salida: Incrementa el radio para alcanzar el perímetro mínimo

Perímetro actual: 26.69

Como se aprecia, el operador morsa permite realizar asignaciones dentro de expresiones, lo que, en muchas ocasiones, permite obtener un código más compacto. Sería conveniente encontrar un equilibrio entre la expresividad y la legibilidad.

3.4 Otras características de las condiciones

Como hemos visto, la sentencia **if**, evalúa una **condición** y ejecuta un bloque u otro en consecuencia.

Python nos ofrece un gran número de posibilidades a la hora de construir condiciones que no solo se emplean en sentencias condicionales, sino que también, como veremos en el siguiente tema, se pueden aplicar perfectamente a un bucle **while**.

En los siguientes apartados, se muestran algunas características interesantes que afectan a las **condiciones en Python**.

3.4.1 Comparación múltiple en una línea

En Python se pueden realizar comparaciones múltiples en una línea sin que la sintaxis deje de ser clara, como vemos a continuación.

- Ejemplo de comparación encadenada

```
x = 5
if 1 < x < 10:
    print("x está entre 1 y 10")
```

Salida: *x está entre 1 y 10*

- Ejemplo de comparación con operadores lógicos

```
x = 7
if x > 5 and x < 10:
    print("x está entre 5 y 10")

# Otra variante con or
y = 3
if y < 5 or y > 10:
    print("y está fuera del rango de 5 a 10")
```

Salida: *x está entre 5 y 10*

y está fuera del rango de 5 a 10

3.4.2 Condición sobre un rango o una lista

En una sentencia condicional **if**, se pueden usar **in** o **not in** para verificar si un valor está en una lista, rango o conjunto.

Los siguientes ejemplos muestra cómo se implementan este tipo de condiciones:

- Comprueba si un valor está dentro de un rango:

```
x = 7

if x in range(5, 10):
    print("x está en el rango de 5 a 10")
```

Salida: *x está en el rango de 5 a 10*

- Comprueba si un valor está contenido en una lista:

```
valores = [1, 2, 3, 4]

if 3 in valores:
    print("3 está en la lista")
```

Salida: *3 está en la lista*

3.4.3 Cortocircuito lógico

Es interesante comprender que las expresiones lógicas no se evalúan por completo si se dan una serie de circunstancias. Aquí es donde entra el concepto de **cortocircuito** que no es más que una forma de denominar a este escenario.

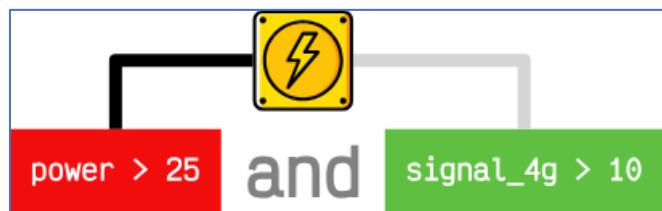
Supongamos un ejemplo en el que utilizamos un **teléfono móvil** que mide la batería por la variable **power** de 0 a 100% y la cobertura 4G por la variable **signal_4g** de 0 a 100%. Para poder enviar un **mensaje por Telegram** necesitamos tener al menos un 25% de batería y al menos un 10% de cobertura:

```
power = 10
signal_4g = 60

if power > 25 and signal_4g > 10:
    print("Se puede enviar mensaje")
else:
    print("NO se puede enviar mensaje")
```

Salida: *NO se puede enviar mensaje*

Dado que estamos en un **and** y la primera condición $power > 25$ no se cumple, se produce un **cortocircuito** y no se sigue evaluando el resto de la expresión porque ya se sabe que va a dar *False*.



Cortocircuito para expresión lógica «and»

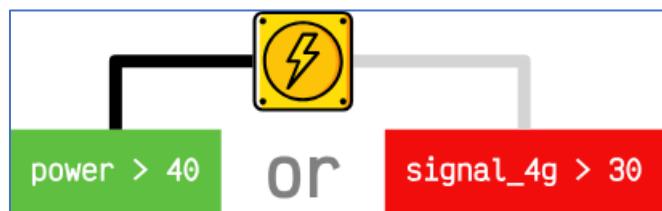
Otro ejemplo. Para poder **hacer una llamada VoIP** necesitamos tener al menos un 40% de batería o al menos un 30% de cobertura:

```
power = 50
signal_4g = 20

if power > 40 or signal_4g > 30:
    print("Se puede hacer llamada")
else:
    print("NO se puede hacer llamada")
```

Salida: *NO se puede hacer llamada*

Dado que estamos en un **or** y la primera condición $power > 40$ se cumple, se produce un **cortocircuito** y no se sigue evaluando el resto de la expresión porque ya se sabe que va a dar *True*.



Cortocircuito para expresión lógica «or»

En ambos casos, si no se produjera un cortocircuito en la evaluación de la expresión, se seguiría comprobando todas las condiciones posteriores hasta llegar al final de la misma.

3.4.4 Booleanos en condiciones

Cuando queremos preguntar por la **veracidad** de una determinada **variable booleana** en una condición, la primera aproximación que parece razonable es la siguiente:

```
is_cold = True

if is_cold == True:
    print('Coge chaqueta')
else:
    print('Usa camiseta')
```

Salida: Coge chaqueta

Aunque lo habitual en Python es *simplificar* esta condición tal que así:

```
is_cold = True

if is_cold:
    print('Coge chaqueta')
else:
    print('Usa camiseta')
```

Salida: Coge chaqueta

Los ejemplos anteriores mostraban una comparación para un valor booleano verdadero (*True*). En el caso de que la comparación fuera para un valor falso (*False*) se debe emplear el operador **not**, como en el siguiente ejemplo:

```
is_cold = False

if not is_cold: # Equivalente a if is_cold == False
    print('Coge camiseta')
else:
    print('Usa chaqueta')
```

Salida: Coge camiseta

De hecho, si lo pensamos, se está reproduciendo bastante bien el *lenguaje natural*:

- Si hace frío, coge chaqueta.
- Si no hace frío, usa camiseta.

3.4.5 Valor nulo. «None»

None es un valor especial de Python que almacena el valor nulo. Veamos cómo se comporta al incorporarlo en condiciones de veracidad:

```
value = None

if value:
    print('Value tiene un valor útil')
else:
    # value podría contener None, False (u otro)
    print('Value parece vacío')
```

Salida: Value parece vacío

Para distinguir *None* de los valores propiamente booleanos, **se recomienda el uso del operador «is»**. Veamos un ejemplo en el que tratamos de averiguar si un valor **es nulo**:

```
value = None

if value is None:
    print('Value es claramente None')
else:
    # value podría contener True, False (u otro)
    print('Value parece vacío')
```

Salida: Value es claramente None

De igual forma, podemos usar esta construcción para el caso contrario. La forma «pitónica» de preguntar si algo **no es nulo** es la siguiente:

```
value = 99

if value is not None:
    print(f'{value=}')
```

Salida: Value=99

Cabe preguntarse por qué utilizamos ***is*** en vez del operador ***==*** al comprobar si un valor es nulo, ya que ambas aproximaciones nos dan el mismo resultado:

```
value = None

if value is None:
    print("True 1")

if value == None:
    print("True 2")
```

Salida: *True 1*

True 2

La respuesta es que el operador ***is*** comprueba únicamente si los identificadores (posiciones en memoria) de dos objetos son iguales siendo una comprobación muy rápida. Sin embargo, la comparación ***==*** puede englobar otras acciones, relacionadas con objetos en las que no vamos a entrar, que hace que su ejecución sea más lenta y que puedan darse «falsos positivos».

Cuando ejecutamos un programa Python existe una serie de objetos precargados en memoria. Uno de ellos es *None*. Podemos ver su identificador usando la función «***id***»¹:

```
print(id(None))
```

Salida: *140737219016656*

Cualquier variable que igualemos al valor nulo, únicamente será una referencia al mismo objeto *None* en memoria:

```
value = None

print(id(value))
```

Salida: *140737219016656*

¹ La función incorporada *id()* recibe como argumento un objeto y retorna otro objeto que sirve como identificador único para el primero. [La función *id\(\)* - Recursos Python](#)

3.4.6 Veracidad

Cuando trabajamos con expresiones que incorporan valores booleanos, se produce una conversión implícita que transforma los tipos de datos involucrados a valores *True* o *False*.

Lo primero que debemos entender de cara comprobar la veracidad son los valores que evalúan a falso o evalúan a verdadero.

A continuación, se muestra todo lo que se evalúa como *False* en Python:

```
bool(False)  
bool(None)  
bool(0)  
bool(0.0)  
bool('') # cadena vacía  
bool([]) # lista vacía  
bool(())) # tupla vacía  
bool({}) # diccionario vacío  
bool(set()) # conjunto vacío
```

El resto de objetos son evaluados como *True*, por ejemplo:

```
bool('False')  
bool(' ')  
bool(1e-10)  
bool([0])  
bool('duck')
```

3.4.7 Asignación lógica

Es posible utilizar operadores lógicos en sentencias de asignación sacando partido de las tablas de la verdad que funcionan para estos casos.

Veamos un ejemplo de asignación lógica utilizando el operador *or*:

```
b = 0
c = 5
a = b or c

print(a)
```

Salida: 5

En la línea “*a = b or c*” se está aplicando una **expresión lógica**, por lo tanto se aplica una conversión implícita de los valores enteros a valores «booleanos». En este sentido el valor 0 se **evalúa a falso** y el valor 5 se evalúa a verdadero. Como estamos en un *or* el resultado será verdadero, que en este caso es el valor 5 asignado finalmente a la variable *a*.

Veamos el **mismo ejemplo de antes**, pero utilizando el operador *and*:

```
b = 0
c = 5
a = b and c

print(a)
```

Salida: 0

En este caso, como estamos en un *and* el resultado será falso, por lo que el valor 0 es asignado finalmente a la variable *a*.

4 Sentencia condicional «match-case»

Como ya se entrevió en el Tema 1, la sentencia **match-case** permite comparar un valor de entrada con una serie de literales. Algo así como un conjunto de sentencias «if» encadenadas.

Veamos un ejemplo que mostramos en el tema anterior:

```
dia = 'viernes'

match dia:
    case 'lunes':
        print('Tengo que hacer la compra')
    case 'martes':
        print('Revisar el trastero')
    case 'miércoles':
        print('Descanso')
    case _:
        print('Día sin planificar')
```

Salida: *Día sin planificar*

Se aprecia como tras la sentencia «*match*» indicamos el valor a evaluar y, en las diferentes sentencias «*case*», indicamos qué valor estamos intentando encontrar. Por último, si se quiere controlar cualquier otro valor que no se haya definido en las sentencias «*case*» se usa el guion bajo “_”:

En este ejemplo veíamos la versión más simple del denominado [pattern matching](#) o búsqueda de patrones y, en los siguientes apartados veremos usos más sofisticado de esta misma sentencia.

4.1 Patrones con tuplas

La sentencia *match-case* va mucho más allá de una simple comparación de valores. Con ella podremos deconstruir estructuras de datos, capturar elementos o mapear valores.

Para exemplificar varias de sus funcionalidades, vamos a partir de una *tupla*² que representará un punto en el plano (2 coordenadas) o en el espacio (3 coordenadas). Lo primero que vamos a hacer es detectar en qué dimensión se encuentra el punto:

```
point = (2, 5)

match point:
    case (x, y):
        print(f'({x},{y}) es un punto en el plano')
    case (x, y, z):
        print(f'({x},{y},{z}) es un punto en el espacio')
```

Salida: (2,5) es un punto en el plano

```
point = (3, 1, 5)

match point:
    case (x, y):
        print(f'({x},{y}) es un punto en el plano')
    case (x, y, z):
        print(f'({x},{y},{z}) es un punto en el espacio')
```

Salida: (3,1,5) es un punto en el espacio

² Una tupla es un listado de valores inmutables. En Python, se representa incluyendo los valores deseados entre paréntesis. Para saber más, podemos consultar Webs como: [Tupla en Python - Aprende cómo crear, acceder y agregar elementos](#)

En cualquier caso, esta aproximación permitiría un punto formado por *strings*:

```
point = ('2', '5')

match point:
    case (x, y):
        print(f'{x}, {y} es un punto en el plano')
    case (x, y, z):
        print(f'{x}, {y}, {z} es un punto en el espacio')
```

Salida: (2,5) es un punto en el plano

Es posible restringir los patrones a un determinado tipo de datos, como es el caso del siguiente ejemplo, en el que se restringen los patrones a valores enteros:

```
point = ('2', '5')

match point:
    case (int(), int()):
        print(f'{point} es un punto en el plano')
    case (int(), int(), int()):
        print(f'{point} es un punto en el espacio')
    case _:
        print('Desconocido!')
```

Salida: Desconocido!

```
point = (3, 1, 5)

match point:
    case (int(), int()):
        print(f'{point} es un punto en el plano')
    case (int(), int(), int()):
        print(f'{point} es un punto en el espacio')
    case _:
        print('Desconocido!')
```

Salida: (3,1,5) es un punto en el espacio

Imaginemos ahora que nos piden calcular la distancia del punto al origen. Debemos tener en cuenta que, a priori, desconocemos si el punto está en el plano o en el espacio:

```
point = (2, 3, 5)

match point:
    case (int(x), int(y)):
        print( (x ** 2 + y ** 2) ** (1 / 2) )
    case (int(x), int(y), int(z)):
        print( (x ** 2 + y ** 2 + z ** 2) ** (1 / 2) )
    case _:
        print('Desconocido!')
```

Salida: 6.164414002968976

De esta forma, se asegura que los puntos de entrada deben tener todas sus coordenadas como valores enteros.

4.2 Patrones con diccionarios

Al igual que en el apartado anterior, explicaremos el uso de patrones en diccionarios utilizando un ejemplo.

Veamos un fragmento de código en el que tenemos que comprobar la estructura de un bloque de autenticación definido mediante un **diccionario**³. Los métodos válidos de autenticación son únicamente dos: bien usando nombre de usuario y contraseña, o bien usando correo electrónico y «token» de acceso. Además, los valores deben venir en formato cadena de texto.

Para solucionar este ejercicio se utilizará un diccionario para los datos (*auths*) que se recorrerá con un bucle *for*⁴:

```
# Lista de diccionarios
auths = [
    {'username': 'sdelquin', 'password': '1234'},
    {'email': 'sdelquin@gmail.com', 'token': '4321'},
    {'email': 'test@test.com', 'password': 'ABCD'},
    {'username': 'sdelquin', 'password': 1234}
]

for auth in auths:
    print(auth)
    match auth:
        case {'username': str(username), 'password': str(password)}:
            print('Autentificación con usuario y contraseña')
            print(f'{username}: {password}')
        case {'email': str(email), 'token': str(token)}:
            print('Autentificación con mail y token')
            print(f'{email}: {token}')
        case _:
            print('Método de autentificación no válido')
    print('---')
```

³ Un diccionario es una colección no ordenada de pares clave-valor. Las claves son únicas e inmutables, mientras que los valores pueden ser de cualquier tipo y pueden ser modificados. Para más información podemos consultar la página [Diccionarios en Python - Blog de Código Facilito](#)

⁴ Estas características del bucle *for* se estudiarán con detenimiento en el próximo tema.

Salida:

```
{'username': 'sdelquin', 'password': '1234'}
```

Autentificación con usuario y contraseña

sdelquin: 1234

```
{'email': 'sdelquin@gmail.com', 'token': '4321'}
```

Autentificación con mail y token

sdelquin@gmail.com: 4321

```
{'email': 'test@test.com', 'password': 'ABCD'}
```

Método de autentificación no válido

```
{'username': 'sdelquin', 'password': 1234}
```

Método de autentificación no válido

4.3 Otras particularidades

Veamos, con un ejemplo, algunas otras particularidades que son aplicable al *pattern matching*. El siguiente código indicará, dada la edad de una persona, si puede beber alcohol:

```
age = 21

match age:
    case 0 | None:
        print('No es una persona')
    case n if n < 17:
        print('No')
    case n if n < 22:
        print('No en USA')
    case _:
        print('Sí')
```

Salida: No en USA

En este código vemos algunas características nuevas:

- En la línea 4 (**case 0 | None:**) se muestra el uso del operador OR «|».
- En las líneas 6 y 8 (**case n if n < 17:**) y (**case n if n < 22:**) se muestra el uso de condiciones dando lugar a *cláusulas guarda*.

4.4 Ejemplos de uso

Este apartado muestra algunos ejemplos de conceptos que se han visto en apartados anteriores con el fin de asentar los conocimientos sobre el uso de la sentencia «*match-case*».

Ejemplo 1.- Usar rangos o agregar condiciones adicionales en el bloque case:

```
x = 7

match x:
    case _ if x < 0:
        print("Número negativo")
    case _ if x == 0:
        print("Es cero")
    case _ if 1 <= x <= 10:
        print("Número entre 1 y 10")
    case _:
        print("Otro número")
```

Salida: Número entre 1 y 10

Ejemplo 2.- Usar patrones de comparación dentro de un rango definido.

```
x = 2

match x:
    case 0:
        print("Es cero")
    case 1 | 2 | 3:
        print("Es uno, dos o tres")
    case _:
        print("No coincide con ningún caso")
```

Salida: Es uno, dos o tres

Ejemplo 3.- Usar patrones para coincidir con tipos específicos.

```
valor = 2
match valor:
    case int():
        print("Es un número entero")
    case str():
        print("Es una cadena")
    case list():
        print("Es una lista")
    case _:
        print("Es otro tipo de dato")
```

Salida: *Es un número entero*

Nota: Si la primera instrucción fuese «`valor = "Hola"`», la salida hubiera sido: *Es una cadena*.

Ejemplo 4.- En objetos complejos como listas o tuplas, se puede desestructurar directamente en el *case*:

```
data = (5, 3)
match data:
    case (x, y) if x > y:
        print(f"x es mayor que y: {x} > {y}")
    case (x, y):
        print(f"Tupla con x={x} y y={y}")
    case _:
        print("No es una tupla válida")
```

Salida: *x es mayor que y: 5 > 3*

5 Web para explorar

Aquí podéis consultar algunas Web con las que podréis profundizar en los concetos vistos en esta Unidad de trabajo.

[El tutorial de Python — documentación de Python - 3.13.0](#)

[Operadores Básicos en Python con ejemplos](#)

[Tupla en Python - Aprende cómo crear, acceder y agregar elementos](#)

[Diccionarios en Python - Blog de Código Facilito](#)