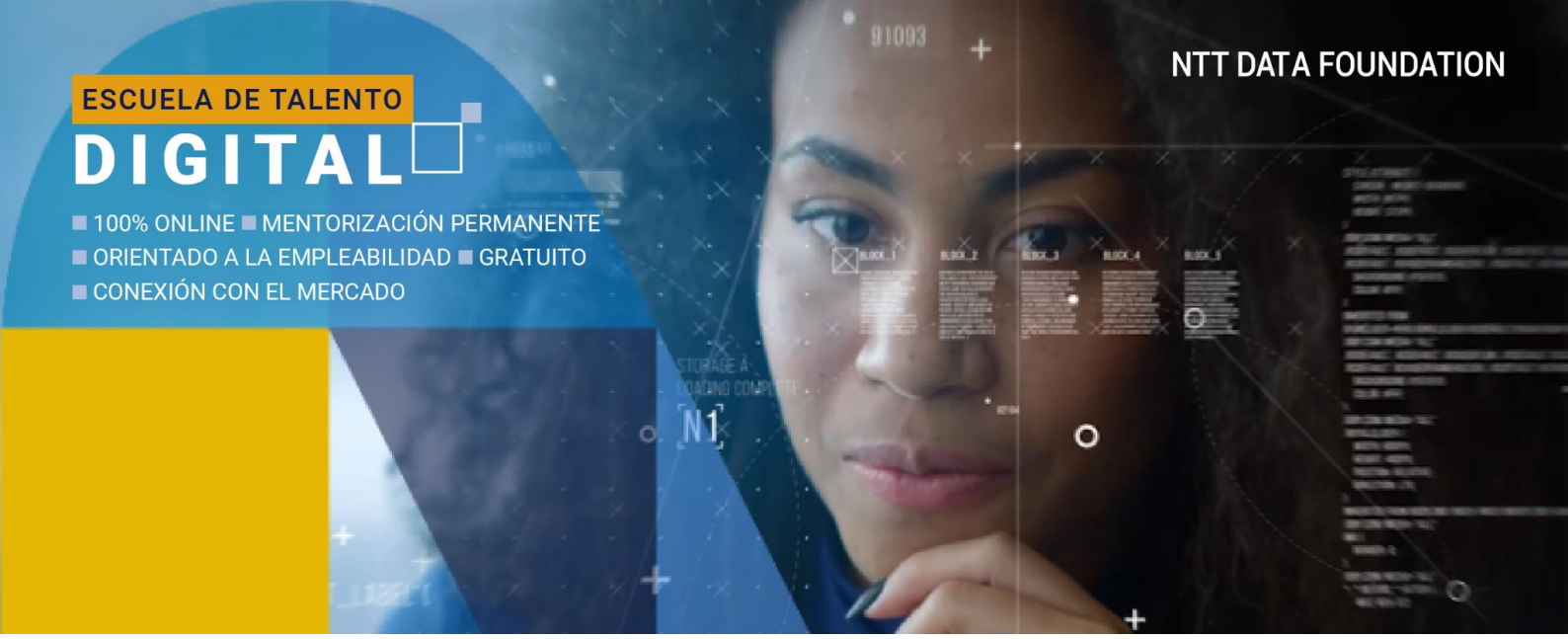

ESCUELA DE TALENTO DIGITAL

NTT DATA FOUNDATION

CONTROL DE FLUJO

ÍNDICE

1. IF - ELIF - ELSE

If es una de las cláusulas condicionales más utilizadas en programación. Esta cláusula **nos permite ejecutar una parte del código u otra**, **en función de que se cumplan una serie de condiciones.**

En lenguaje natural, sería equivalente a decir, si pasa esto, vamos a hacer esto, pero si no pasa esto, vamos a hacer esto otro. Por ejemplo, si llueve, sacaré el paraguas, pero si no llueve, no lo voy a sacar.

1.1. IF

La estructura de una sentencia `if` es:

```
if condición a cumplir:
    código a ejecutar
```

Esto quiere decir que, si se cumple la condición que está después del `if`, entonces se ejecutará el código que está debajo del `if`.

Con el ejemplo del paraguas, en lenguaje natural, sería el equivalente a decir:

si llueve:

    saco el paraguas

Otro ejemplo, ya utilizando la sentencia en formato código, sería:

```
if (a > b):
    print(“Es mayor”)
```

Esto quiere decir que si el valor que tiene la variable “a” es mayor que el valor que tiene la variable “b”, entonces se mostrará por pantalla el mensaje “Es mayor”.

1.2. ELSE

Pero ¿qué es lo que ocurre si no se cumple la condición?

Pues, si no hay ninguna sentencia posterior a `if`, simplemente no se ejecutará el código después de `if`, y ya está, no pasará nada más. Pero, normalmente, si se cumple una condición queremos que el programa haga una cosa y, si no se cumple, queremos que haga otra. Son dos caminos, hay una bifurcación, si pasa esto, harás esto y si pasa esto otro, harás esto otro.

En programación esto se hace utilizando la sentencia `else`. Y se utiliza de la siguiente manera:

```
if condición a cumplir:
      ejecutar este código
else:
      ejecutar este otro código
```

Con el ejemplo del paraguas, en lenguaje natural, sería el equivalente a decir:

si llueve:

    saco el paraguas

pero si no llueve:

    no saco el paraguas

Continuando con el ejemplo de código anterior, podríamos tener lo siguiente:

```
if (a > b):
      print("Es mayor")
else:
      print("Es menor")
```

Esto quiere decir que, si el valor que tiene la variable "a" es mayor que el valor que tiene la variable "b", entonces se mostrará por pantalla el mensaje "Es mayor", pero si esto no se cumple, es decir, que el valor de "a" es menor que el valor de "b", se mostrará por pantalla "Es menor".

En ocasiones, podemos necesitar comprobar más de una condición que se pueden combinar en la misma sentencia `if` utilizando operadores. Por ejemplo, siguiendo con el ejemplo anterior, si tenemos tres variables "a", "b" y "c", y queremos saber si `a>b` y si `b>c`, podemos tener el siguiente código:

```
if (a > b) and (b > c):
      print("Es mayor")
else:
      print("Es menor")
```

En este caso, si se cumplen las dos condiciones (a > b) y (b > c) entonces, aparecerá en pantalla "Es mayor", si alguna de las dos no se cumple, aparecerá en pantalla "Es menor", porque, recordad que, el operador "and" devolvía "verdadero" siempre que las dos partes de la sentencia fueran verdaderas.

Sin embargo, si tenemos este código:

```
if (a > b) or (b > c):
```

```
        print("Es mayor")
else:
        print("Es menor")
```

En este caso, si se cumple cualquiera de las dos condiciones, y solo una de ellas, o bien (a > b) o (b > c), se mostrará en pantalla "Es mayor". De ahí la importancia de saber qué queremos realmente que se cumpla y de utilizar los operadores correctos.

1.3. ELIF

Cuando queremos comprobar más de una condición podemos utilizar la sentencia `elif`, que es el abreviado de `else if`, es decir, se podría expresar como "si no se cumple la primera condición pero sí se cumple esta otra condición entonces haremos ….".

En código, esto se expresaría de la siguiente manera:

```
if condición 1 a cumplir:
        ejecutar este código
elif condición 2 a cumplir:
        ejecutar este código
else:
        ejecutar este otro código
```

Con el ejemplo del paraguas, en lenguaje natural, sería el equivalente a decir:

si llueve:

        saco el paraguas

pero si no llueve, y el tiempo da probabilidad de lluvia:

        también me llevo el paraguas

pero si no llueve y el tiempo tampoco da probabilidad de lluvia:

        no saco el paraguas

Continuando con el ejemplo de código anterior, podríamos tener lo siguiente:

```
if (a > b):
        print("Es mayor")
elif (a == b):
        print("Es igual")
```

```
else:
      print(“Es menor”)
```

Esto quiere decir que, si el valor que tiene la variable “a” es mayor que el valor que tiene la variable “b”, entonces se mostrará por pantalla el mensaje “Es mayor”, pero si esto no se cumple, y se cumple que el valor que tiene la variable “a” es igual al valor que tiene la variable “b”, entonces se mostrará por pantalla el mensaje “Es igual”, pero si ninguna de las dos condiciones anteriores se cumple, entonces se mostrará “Es menor”.

2. IF ANIDADOS

La sentencia `if` también puede contener otras sentencias `if` en su código a ejecutar, es decir, una sentencia `if` puede tener anidada otras sentencias `if`.

Si queremos ejecutar un código en el que, si se cumple una condición, y a su vez, si se cumple otra condición después de haberse cumplido la primera se ejecute otro código, se anidarán dos `if`. Veámoslo con un ejemplo. Si tenemos el código:

```
fruta = True

tipo = “naranja”

if fruta == True:
      if tipo == “naranja”:
            print(“Tengo una naranja”)
      else:
            print(“Tengo una fruta que no es una naranja”)
else:
      print(“No tengo una fruta”)
```

Esto quiere decir que, si tengo una fruta (si `fruta == True`) y si esta fruta es de tipo naranja (si `tipo == “naranja”`), entonces el programa me mostrará `“Tengo una naranja”`; pero si tengo una fruta (`fruta == True`) que no es una naranja (`tipo != naranja`), entonces el programa mostrará `“Tengo una fruta que no es una naranja”`; por último, si no tengo una fruta (`fruta == False`), el programa mostrará `“No tengo una fruta”`.

En este punto, te podrías preguntar, ¿por qué no se utiliza la sentencia `if fruta == True and tipo == “naranja”` y se usa un solo `if`?

En ese caso, el código que podrías crear sería:

```
fruta = True

tipo = “naranja”
```

```
if fruta == True and tipo == “naranja”:
      print(“Tengo una naranja”)
else:
      print(“No tengo una fruta”)
```

Pero no podrías lanzar el mensaje `“Tengo una fruta que no es una naranja”`, que sucede solo cuando tienes una fruta, pero tipo no es igual a naranja.

Estas cláusulas anidadas se utilizan cuando tienes que dar diferentes opciones, si se cumplen o no diferentes condiciones.

3. WHILE

While es una de las cláusulas que se utiliza en programación para ejecutar bucles. Los bucles son elementos que nos permiten ejecutar operaciones de forma repetitiva siempre que se cumpla, o deje de cumplirse, una condición.

La cláusula `while` nos permite ejecutar código mientras se cumpla una condición dada.

En lenguaje natural, sería equivalente a decir, mientras se cumpla esto, haz esto, pero cuando deje de cumplirse, deja de hacerlo. Por ejemplo, mientras esté trabajando, no me interrumpas, pero cuando deje de trabajar ya me puedes interrumpir.

La estructura de una sentencia `while` es:

```
while condición a cumplir:
      código a ejecutar
```

Esto quiere decir que, si se cumple la condición que está después del `while`, entonces se ejecutará el código que está debajo del `while`. Y esto se va a ejecutar continuamente mientras se cumpla la condición.

Con el ejemplo en lenguaje natural, sería el equivalente a decir:

mientras esté trabajando:

    no me interrumpas

Otro ejemplo, ya utilizando la sentencia en formato código, sería:

```
x = 1
while x < 5:
      print(str(x) + “ es menor que 5”)
```

Esto quiere decir que, mientras el valor que tiene la variable “x” sea menor que 5, entonces se mostrará por pantalla el mensaje `“x es menor que 5”`, sustituyéndose la “x” por el valor

que tenga la variable en ese momento, en nuestro caso, ahora mismo, la “x” valdría 1, por lo que el texto sería `“1 es menor que 5”`.

Pero, el código anterior tendría un problema y es que, como el valor de “x” no varía nunca, siempre es 1, este bucle `while` se estaría ejecutando siempre, porque siempre se cumpliría la condición, por lo que tendríamos que añadir una sentencia más que vaya modificando el valor de esa “x”. Por ejemplo:

```
x = 1

while x < 5:

      print(str(x) + “ es menor que 5”)

      x = x + 1 (o x += 1, que haría lo mismo)
```

Ahora sí, esta última sentencia, suma a “x” uno, cada vez que se ejecuta el código, funcionando de la siguiente manera:

- La primera vez que se ejecuta la condición de `while` valora si `x < 5`, como x vale 1, 1 < 5 es verdadero y, por lo tanto, el programa muestra `“1 es menor que 5”`, y a “x” le suma uno.
- Ahora cuando se ejecuta de nuevo, la condición `while` valora si `x < 5`, como ahora “x” vale 2, y 2 < 5 es verdadero, el programa muestra `“2 es menor que 5”`, y a “x” le suma uno.
- Así seguiría ejecutándose el bucle hasta que `x = 5`, en ese caso la sentencia `while` valora que x < 5, como 5 < 5 es falso, entonces el bucle termina y no se muestra ningún resultado más, con lo que el resultado final será:

  ```
  1 es menor que 5
  2 es menor que 5
  3 es menor que 5
  4 es menor que 5
  ```

3.1. Sentencia Break

Break es una sentencia que nos va a permitir salir de un bucle sin tener que esperar a que se ejecuten todas las repeticiones. Esta sentencia se suele utilizar cuando tenemos una lista de elementos muy grandes y queremos que el bucle finalice cuando encontremos un elemento específico sin tener que mirar toda la lista.

También podemos utilizar esta sentencia en situaciones como la anterior, cuando no se modificaba el valor de una variable, para que el bucle no se ejecute infinitas veces.

Se utiliza de la siguiente manera:

```
while condición a cumplir:

      ejecutar código

      break
```

En el caso del ejemplo anterior, podríamos tener esto:

```
x = 1

while x < 5:

      print(str(x) + “ es menor que 5”)

break
```

En este caso, como “x” es igual a 1, entontes se cumple la condición 1 < 5, por lo que se escribe la sentencia `“1 es menor que 5”`, y al encontrarnos con la sentencia `break`, salimos del bucle y no hacemos nada más.

Con otro ejemplo, podríamos tener:

```
x = 1

while x < 200:

      print(str(x) + “ es menor que 200”)

      if x == 3:

            print(“Salimos del bucle”)

            break

      x = x+1
```

En este caso, no listamos de 1 a 200, sino que cuando “x” es igual a 3, entramos en la cláusula `if`, y ejecutamos el `break` después de sacar por pantalla `“Salimos del bucle”`, por lo que nuestro resultado será:

```
1 es menor que 200

2 es menor que 200

3 es menor que 200

Salimos del bucle
```

3.2. Sentencia Continue

Otra sentencia que se suele utilizar con los bucles while es la sentencia **continue**. Esta sentencia se utiliza para saltarse una iteración, o una repetición, del bucle. La diferencia con la sentencia break es que break rompe el bucle, lo termina, sin embargo, continue salta una iteración, haciendo que todo lo que está escrito detrás no se ejecute, pero sí vuelva a la siguiente iteración, si aún se cumple la condición.

Veamos cómo funciona directamente con un ejemplo. Si tenemos el siguiente código:

```
x = 0
```

```
while x < 5:
    x = x + 1
    if x == 3:
        continue
    print(str(x) + " es menor o igual que 5")
```

En este caso, incrementamos el valor de “x” nada más entrar al bucle porque si lo hacemos después de continue se saltará esa orden y volverá a `while` una y otra vez sin que el bucle nunca pare porque “x” siempre será 0 y, por lo tanto, menor que 5.

Entonces, entramos en el bucle porque 0 < 5 es verdadero, sumamos 1 a “x”, con lo que “x” valdrá 1, como es distinto de 3 sacará por pantalla `“1 es menor o igual que 5”`. En la segunda iteración hará lo mismo, sacando por pantalla `“2 es menor o igual que 5”`.

En la tercera iteración “x” ya valdrá 3, por lo que entrará en el `if` al cumplirse la condición y ejecutará el `continue`. Esto lo que hará es que vuelva al `while` sin tener en cuenta todo lo que hay después, en nuestro caso el `print`, por lo que la sentencia `“3 es menor o igual que 5”` no se imprimirá.

Entonces, al volver al `while` valora la condición, como “x” vale 3, y es menor que 5, entonces le sumará 1, al valer 4 ya no entra en el `if` y saca `“4 es menor o igual que 5”`. Vuelve al `while` y como 4 < 5, le suma 1 y, como no es igual a 3, escribe `“5 es menor o igual que 5”`. Vuelve al `while` y, en este caso, como 5 < 5 es `False`, el bucle finaliza.

Con esto nuestro resultado será:

```
1 es menor o igual que 5
2 es menor o igual que 5
4 es menor o igual que 5
5 es menor o igual que 5
```

4. FOR

For es otra de las cláusulas que se utiliza en programación para ejecutar bucles. La diferencia con while es que for realiza repeticiones sobre listas, diccionarios o conjuntos que ya tienen predefinidos el número de elementos que contienen y, por lo tanto, el número de iteraciones que se van a realizar.

En lenguaje natural, sería equivalente a decir, para este elemento, que va recorriendo esta lista, haz esto, pero cuando se acabe la lista, deja de hacerlo. Por ejemplo, mira todos los nombres que hay en esta lista y dime cuáles son compuestos. Esto mirará uno por uno los elementos que tenga la lista y ejecutará la orden de devolver los que sean compuestos.

La estructura de una sentencia `for` es:

```
for elemento in lista:

      código a ejecutar
```

Aquí, elemento es realmente un índice que recorre la lista y ejecuta el código mientras que haya elementos en la lista.

Un ejemplo utilizando la sentencia en formato código, sería:

```
lista = [“perro”, “gato”, “pájaro”]

for i in lista:

      print(“Animal:”, i)
```

Esto funcionaría de la siguiente manera:

- “i” toma, en primer lugar, el primer valor de lista, es decir, `i = “perro”`, entonces, el programa entendería que para `i = “perro”` escribe `“Animal: perro”`, y mostraría esto por pantalla.
- En la segunda iteración, “i” tomaría el segundo valor de la lista, en este caso, `i = “gato”`, y sacaría por pantalla `“Animal: gato”`.
- En la tercera iteración, “i” tomaría el tercer valor de la lista, en este caso, `i = “pájaro”`, y sacaría por pantalla `“Animal: pájaro”`.
- Al llegar a la cuarta iteración, como no hay un cuarto elemento en la lista, el bucle finaliza.
- El resultado que obtendríamos sería:
  ```
  Animal: perro
  Animal: gato
  Animal: pájaro
  ```

En el caso de los bucles `for`, la variable “i” se va incrementando sumando 1 a su valor de forma automática, no necesitamos indicarle al programa que le sume 1, como sucedía en los bucles `while`.

4.1. Range

Range es una sentencia que nos permite indicar a for cuál es el número de veces que queremos que se realicen las iteraciones. Esto se utiliza cuando sabemos, exactamente, el número de veces que queremos que se ejecute un código, sin necesidad de tener una lista, conjunto o diccionario.

Por ejemplo, si queremos mostrar por pantalla tres veces la misma frase, tendríamos este código:

```
for i in range (0, 3):

      print(“Esta es la frase que quiere mostrar tres veces.”)
```

Esto nos mostraría por pantalla:

```
Esta es la frase que quiere mostrar tres veces.
Esta es la frase que quiere mostrar tres veces.
Esta es la frase que quiere mostrar tres veces.
```

¿Cómo funciona? Sería lo mismo que decir, para "i", desde 0 hasta 3 (valor de `i = 0`, valor de `i = 1` y valor de `i = 2`), ejecuta la acción que te indico. Es decir, la primera vez que llegamos al `for`, que es la primera iteración, el valor de "i" es 0 y se ejecuta la acción; volvemos al `for`, para la segunda iteración, donde `i = 1`, y se ejecuta la acción; entonces, volvemos al `for`, para la tercera iteración, donde `i = 2`, y se ejecuta la acción; y al ser la tercera iteración, y en `range` le hemos dicho que queremos 3 iteraciones, finaliza el bucle y no se realizan más iteraciones.

En este caso pasa lo mismo que lo que pasa cuando recorremos cadenas de caracteres, que el índice siempre empieza en 0, y lo deberemos tener en cuenta a lo hora de incluir el valor inicial. También debemos tener en cuenta que el segundo índice siempre será uno más del valor que queremos que tenga nuestro índice "i".

Realmente el código interpreta internamente el valor del segundo parámetro de `range`, en este caso el 3, como el valor que quiero que tenga mi índice más 1, es decir, el valor que quiero que tenga mi índice será 3 – 1 -> `i = 2`.

Es decir, si tenemos, por ejemplo, el siguiente código:

```
for i range (1, 3):

      print("Esta es la frase que quiero repetir.")
```

El resultado será:

```
Esta es la frase que quiero repetir.
Esta es la frase que quiero repetir.
```

Porque el bucle ejecutará la acción cuando `i = 1` y cuando `i = 2`, porque la primera iteración será `i = 0`, pero no la tendrá en cuenta. Es decir, funcionaría así:

- `i = 0`, como no está en el intervalo `(1, 3)`, no lo ejecuto, e incremento el valor de i.
- `i = 1`, como está en el intervalo `(1, 3)`, es decir, no es todavía igual a 3, lo ejecuto e incremento el valor de i.
- `i = 2`, como está en el intervalo `(1, 3)`, y aún no es igual a 3, lo ejecuto e incremento el valor de i.
- `i = 3`, como el valor de "i" ya es tres, no lo ejecuto y finalizo el bucle.

Entonces, si queremos empezar en 1 y queremos que se repita tres veces, el segundo parámetro de `range` deberá ser 4.

```
for i range (1, 4):

      print("Esta es la frase que quiero repetir.")
```

El resultado será:

```
Esta es la frase que quiero repetir.
Esta es la frase que quiero repetir.
Esta es la frase que quiero repetir.
```

Funcionando de esta manera:

- `i = 0`, como no está en el intervalo `(1, 4)`, no lo ejecuto, e incremento el valor de i.
- `i = 1`, como está en el intervalo `(1, 4)`, y aún no es igual a 4, lo ejecuto e incremento el valor de i.
- `i = 2`, como está en el intervalo `(1, 4)`, y aún no es igual a 4, lo ejecuto e incremento el valor de i.
- `i = 3`, como está en el intervalo `(1, 4)`, y aún no es igual a 4, lo ejecuto e incremento el valor de i.
- `i = 4`, como el valor de "i" ya es cuatro, no lo ejecuto y finalizo el bucle.

4.2. Break y Continue

Las sentencias break y continue funcionan exactamente igual que en los bucles while.

Break nos va a permitir interrumpir la ejecución del bucle for en un momento dado. Por ejemplo:

```
lista = [“perro”, “gato”, “pájaro”, “ratón”]

for i in lista:

      if i == “pájaro”:

            break

      print(“Animal:”, i)
```

Nos devolverá como resultado:

```
Animal: perro
Animal: gato
```

Porque al llegar al punto en el que `i = “pájaro”` el bucle entra en el `if` y se ejecuta el `break` deteniendo el bucle.

Continue nos va a permitir saltarnos una iteración del bucle for. Por ejemplo:

```
lista = [“perro”, “gato”, “pájaro”, “ratón”]

for i in lista:

      if i == “pájaro”:

            continue

      print(“Animal:”, i)
```

Nos devolverá como resultado:

```
Animal: perro
Animal: gato
Animal: ratón
```

Ya que cuando `i = “pájaro”` entramos en el `if`, ejecutamos continue y volvemos al bucle sin terminar de ejecutar el código que hay por debajo.

4.3. Bucles anidados

En ocasiones podemos necesitar anidar más de un bucle, es decir, podemos necesitar ejecutar un bucle dentro de otro bucle. Esto es posible tanto con while como con for.

Veámoslo con ejemplos.

Si queremos combinar dos listas de elementos, podemos tener el siguiente código:

```
lista1 = [“perro”, “gato”, “pájaro”, “ratón”]

lista2 = [“blanco”, “negro”]

for x in lista1:

      for i in lista2:

            print(f”Animal: {x} {i})
```

Con lo que tendríamos como resultado:

```
Animal: perro blanco
Animal: perro negro
Animal: gato blanco
Animal: gato negro
Animal: pájaro blanco
Animal: pájaro negro
Animal: ratón blanco
Animal: ratón negro
```

Esto funciona de la siguiente manera:

- `x = “perro”` en el primer `for`, voy al segundo `for` y `i = “blanco”`, entonces ejecuto el código dentro del segundo `for`, saco por pantalla `“Animal: perro blanco”`, e incremento la i.
- En este punto, no he salido aún del segundo bucle, así que todavía no he incrementado el valor de “x”. Por lo tanto, `i = “negro”`, pero `x = “perro”` aún. Por lo que ejecuto el código, saco por pantalla: `“Animal: perro negro”`, e incremento el valor de “i”. Como se ha terminado la lista2, vuelvo al primer bucle e incremento el valor de “x”.

- Ahora `x = “gato”`, voy al segundo `for` y es como si no se hubiera ejecutado nunca y se reinicia, por lo que `i = “blanco”` de nuevo, y se vuelve a recorrer la lista. En este momento, el valor de “i” es válido, ejecuto el código, saco por pantalla `“Animal: gato blanco”` e incremento el valor de “i”, repitiendo el proceso.
- Esto se repetirá tantas veces como valores pueda tomar “x”, por lo que tendremos el resultado indicado anteriormente.

Si hacemos una modificación en el ejemplo anterior, y recorremos primero la lista2 y después la lista1, tendremos lo siguiente:

```
lista1 = [“perro”, “gato”, “pájaro”, “ratón”]

lista2 = [“blanco”, “negro”]

for x in lista2:

      for i in lista1:

            print(f”Animal: {x} {i})
```

Y como resultado:

```
Animal: blanco perro
Animal: blanco gato
Animal: blanco pájaro
Animal: blanco ratón
Animal: negro perro
Animal: negro gato
Animal: negro pájaro
Animal: negro ratón
```

Igual que en el caso anterior, aquí empieza recorriendo primero la lista2, por lo que `x = “blanco”`, y luego recorrerá la lista1, por lo que `i = “perro”`, y seguirá recorriendo la lista1, la del segundo bucle, hasta que `i = “ratón”`. Cuando termine, volverá al primer `for`, y al siguiente elemento de la lista2, en este caso `x = “negro”`, y volverá a recorrer completa la lista1.

En el caso de bucles `while` pasa lo mismo. Si tenemos el código:

```
i = 0
j = 0

while i < 2:

    while j < 3:

        print(i,j)

        j = j + 1

    i = i + 1
```

```
    j = 0
```

En este caso el programa funcionaría así:

- En el primer bucle `while` evaluaría la condición `i < 2`, en nuestro caso, 0 < 2, como es verdadero entraría en el bucle.
- Llegaría al segundo `while` y evaluaría la condición `j < 3`, en nuestro caso, 0 < 3, como es verdadero entraría en este segundo bucle.
- Entonces ejecutaría el código, sacaría por pantalla 0 0, y a j le sumaría 1, con lo que `j = 1`.
- Ahora, evaluaría la condición de este segundo bucle, `j < 3`, para ver si lo vuelve a ejecutar. Como 1 < 3 es verdadero, saca por pantalla 0 1, y a j le suma 1.
- Esto funcionaría así hasta que `j = 3`.
- Cuando j es igual 3, como 3 < 3 es falso, no ejecuta el segundo bucle y pasa directamente a ejecutar `i = i +1`, pasando i a valer 1, y ejecutaría la sentencia `j = 0`, volviendo j a tener ese valor.
- Entonces, evaluaría la condición del primer bucle, `i < 2`, como 1 < 2 es verdadero, volvería a evaluar la condición del segundo bucle. Como j = 0 de nuevo, lo volverá a ejecutar como antes hasta que lleguemos a que j = 3 y como 3 < 3 es falso, no se ejecuta y suma 1 a i y vuelve a dar a j el valor 0.
- Como ahora `i = 2` y 2 < 2 es falso, no entra en el primer bucle y se termina el programa, mostrando como resultado:

```
0 0
0 1
0 2
1 0
1 1
1 2
```

5. EXPRESIONES ANIDADAS

Las expresiones anidadas son expresiones que se utilizan dentro de otras expresiones.

Estas expresiones suelen ser expresiones matemáticas que se pueden utilizar, por ejemplo, dentro de la función `print`, en la función `print` combinadas con condicionales, como condición en cláusulas condicionales o bucles, etc.

Por ejemplo, dentro de una función `print` podríamos tener:

```
print(“El resultado es”, (1+7)*3/2)
```

Lo que nos devolvería:

```
El resultado es 12
```

En la función `print` combinada con un condicional, tendríamos, por ejemplo:

```
a = 2
```

```
b = 4

print(“El valor más pequeño entre a y b es”, a if a < b else b)
```

Este código sería el mismo que este:

```
a = 2

b = 4

if a < b:

      print(“El valor más pequeño entre a y b es”, a)

else:

      print(“El valor más pequeño entre a y b es”, b)
```

Pero, como veis, para el primero solo necesito tres líneas de código y para el segundo seis.

Como condición en un bucle podríamos tener:

```
a = int(input("Introduce un valor para a: "))

while (3 + a) / 10 < 10:

        print("El valor de a es", a)

        a += 1

print("La operación es menor que 10")
```

Donde se imprimiría “El valor de a es xx” en función del valor que le demos a “a”, tantas veces hasta que deje de cumplirse que el valor de la operación sea menor que 10.

5.1. Reglas de precedencia

Uno de los puntos más importantes que debemos tener en cuenta a la hora de utilizar expresiones anidadas son las reglas de precedencia. Estas reglas determinan cuál es el orden en el que se deben evaluar las diferentes expresiones anidadas, es decir, qué operaciones se realizan primero y como se combinan.

El orden que determinan las reglas de precedencia es:

1. Expresiones entre paréntesis. Cualquier expresión que esté entre paréntesis se evaluará siempre en primer lugar.
2. Expresiones aritméticas: +, -, *, /, %.
3. Expresiones relacionales: <, >, ==, != <=, >=.
4. Expresiones lógicas: and, or y not.

También hay que tener en cuenta que entre las expresiones aritméticas se utiliza el orden de matemáticas, es decir, primero multiplicaciones y divisiones y, después, sumas y restas.

Esto qué quiere decir, que no devuelve el mismo resultado la operación `1+7*3/2`, cuyo resultado es `11.5`, que `(1+7)*3/2`, cuyo resultado es `12`.

En la primera expresión, el orden del cálculo sería `7*3 = 21 / 2 = 10.5 +1 = 11.5`, sin embargo, en la segunda expresión el orden sería `(1 + 7) = 8 * 3 = 24 / 2 = 12`.

Otro ejemplo, utilizando otro tipo de expresiones, podría ser:

`2 + 4 / 2 >= 4 or (2 + 4) / 2 < 6`, cuyo resultado sería `True`, o esta otra:

`2 + 4 / 2 >= 4 and (2 + 4) / 2 < 3`, cuyo resultado sería `False`.

Si la incluimos en un `print`, con un `if`, podríamos tener:

`print(True if 2 + 4 / 2 >= 4 and (2 + 4) / 2 < 3 == True else False)` que devolvería `False`, o

`print(True if 2 + 4 / 2 >= 4 or (2 + 4) / 2 < 6 == True else False)` que devolvería `True.`

6. MÁS INFORMACIÓN

Aquí puedes ver algunos ejemplos de ejercicios resueltos que usan la sentencia condicional if: https://www.youtube.com/watch?v=PKFKoAN2zEo

Aquí puedes ver un ejemplo de cómo se ejecutaría un programa con un bucle while, analizando por dónde se va ejecutando y cuáles son los valores, paso a paso: https://www.youtube.com/watch?v=vhW_cIidSQI&list=PLb_E6BNMg5j7-MJ0ctjvKQlv2PU7qbMDb&index=23

Aquí puedes ver algunos ejemplos de ejercicios resueltos que usan la sentencia while: https://www.youtube.com/watch?v=I6T_qjYiDDM&list=PLb_E6BNMg5j7-MJ0ctjvKQlv2PU7qbMDb&index=22&t=1368s

Aquí puedes ver algunos ejemplos de ejercicios resueltos que usan la sentencia for: https://www.youtube.com/watch?v=cmFX38TpxNM&list=PLb_E6BNMg5j7-MJ0ctjvKQlv2PU7qbMDb&index=19&t=252s

Aquí puedes ver un ejemplo de un ejercicio que anida bucles for: https://www.youtube.com/watch?v=7fBMgfbD570&list=PLb_E6BNMg5j7-MJ0ctjvKQlv2PU7qbMDb&index=20

Aquí puedes ver unos ejemplos de cómo se usan break y continue que también incluyen un for y un while anidados: https://www.youtube.com/watch?v=lG-DTUOZVZg&list=PLb_E6BNMg5j7-MJ0ctjvKQlv2PU7qbMDb&index=26&t=58s

En estos dos vídeos puedes ver algunos ejemplos sobre precedencia, que será muy útil a la hora de programar expresiones anidadas:

Vídeo 1: https://www.youtube.com/watch?v=VbbEeFtc__g

Vídeo 2: https://www.youtube.com/watch?v=4P0T3jWf748