TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE

- ¿Para qué sirven los ciclos?
- Creando un ciclo con while
- Instrucción For
- Evaluando condiciones en ciclos
- Ciclos infinitos
- Utilizando Ciclos anidados
- Combinación de ciclos con instrucciones if/else
- Utilizando las sentencias break y continue

¿PARA QUÉ SIRVEN LOS CICLOS?

Los ciclos son un elemento de programación que repite un número determinado de veces una porción de código, hasta completar el proceso deseado. Las tareas repetitivas son comunes en la programación, y los ciclos son esenciales para ahorrar tiempo y minimizar los errores.

En un ejemplo de la vida real, podríamos escribir los pasos para hacer un sándwich de jamón y queso. ¿Qué pasa si tenemos que hacer 500 sándwiches para una excursión escolar? En lugar de tener que hacer los mismos pasos una y otra vez, probablemente preferiríamos hacerlo una sola vez, y que se repitiera automáticamente hasta completar los 500 sándwiches.

Cuando los programadores escriben código, los ciclos les permiten acortar lo que podrían ser cientos de líneas de código a solo unas pocas. Esto les deja escribir el código una vez, y repetirlo tantas veces como sea necesario.

Los ciclos hacen que el código sea más manejable y organizado.

CREANDO UN CICLO CON WHILE

Recordemos la sintaxis de la sentencia **while:**

```
while expresión_de_prueba:
    Cuerpo del while
```

Usualmente las expresiones de prueba son sencillas, haciendo uso de contadores que limitan la cantidad de repeticiones del ciclo a un número determinado, o también de banderas que se evalúan y actualizan en cada bucle. Sin embargo, las expresiones de prueba pueden ser tan complejas como lo requiera el caso. Para ello podemos tener múltiples condiciones que cumplir.

Su sintaxis utilizando el operador **AND**, sería la siguiente:

```
while expresión_de_prueba1 and expresión_de_prueba2 … :
    Cuerpo del while
```

Si utilizamos el operador **OR**, la sintaxis es:

```
while expresión_de_prueba1 or expresión_de_prueba2 … :
    Cuerpo del while
```

Los ciclos **while** también pueden tener un bloque **else** opcional. La parte **else** se ejecuta si la condición del bucle **while** se evalúa como *False*.

INSTRUCCIÓN FOR

Recordemos la sintaxis de la expresión **for**:

```
for elem in secuencia:
        instrucciones del bucle
```

En el módulo 2 vimos un ejemplo sobre cómo utilizar el **for** en una lista, y ahora cómo usarlo con otras estructuras de datos.

Independientemente de las diferencias entre las listas y las tuplas, el bucle sobre estas estructuras de datos es muy similar.

```
x = (10, 20, 30, 40, 50)
for var in x:
    print("Índice "+ str(x.index(var)) + ":",var)

#Resultado:

Índice 0: 10
Índice 1: 20
Índice 2: 30
Índice 3: 40
Índice 4: 50
```

Si tenemos una lista de tuplas, podemos acceder a los elementos individuales de cada tupla de nuestra lista incluyendo ambos como variables en el bucle **for**, de esta manera:

```
x = [(1,2), (3,4), (5,6)]

for a, b in x:
    print(a, "plus", b, "equals", a+b)
```

Además de las listas y las tuplas, los diccionarios son otro tipo de datos de Python que es probable que encuentres al trabajar con datos, y los bucles **for** también pueden iterar a través de los diccionarios.

Los diccionarios de Python se componen de pares clave-valor, por lo que, en cada bucle, hay dos elementos a los que necesitamos acceder (la clave y el valor). Para recorrer tanto las claves, como los valores correspondientes a cada par clave-valor, necesitamos llamar al método `.items().`

Los bucles **for** también pueden iterar a través de cada carácter de una cadena. Así es como funciona:

```
print("Ciencia")
for caracter in "Ciencia ":
    print(caracter)

#Resultado:
Ciencia
C
i
e
n
c
i
a
```

Un bucle **for** puede tener también un bloque **else** opcional. La parte **else** se ejecuta al agotarse los elementos de la secuencia en el bucle **for**.

```
digitos = [0, 1, 5]

for i in digitos:
    print(i)
else:
    print("No quedan elementos en la lista.")

#Resultado:
0
1
```

```
5
No quedan elementos en la lista.
```

EVALUANDO CONDICIONES EN CICLOS

Cuando escribimos las condiciones de un ciclo, éstas pueden ser tan complejas como se requiera, puede tener N cantidad de subcondiciones, y puede tener una mezcla de condicionales (**if**, **or**, **not**), simplemente debemos tomar en cuenta cómo se evalúan las condiciones en estos casos.

Las condiciones se van evaluando en el orden en que aparecen escritas, de esta manera si la primera condición es falsa, se considera que toda la expresión es falsa y el ciclo se detiene; y en caso de ser verdadera, se evalúa la próxima condición tomando en cuenta el operador lógico (**and**, **or** y **not**).

Si la segunda condición es falsa, tomando en cuenta el operador lógico entre ellas, se considera que toda la expresión es falsa y se detiene el ciclo; en caso de ser verdadera, se evalúa la próxima condición tomando en cuenta el operador lógico.

Así sucesivamente, hasta llegar a la última condición, y en caso de que esta sea verdadera se ejecuta el cuerpo del ciclo.

```
 a = 5
 b = 10
iteraciones = 5

cuenta = 0
while cuenta < a or cuenta < b and not cuenta >= iteraciones:
print("Cuenta: { cuenta }, a: {a}, b: {b}")
cuenta += 1

#Resultado:
Cuenta: 0, a: 5, b: 10
Cuenta: 1, a: 5, b: 10
Cuenta: 2, a: 5, b: 10
Cuenta: 3, a: 5, b: 10
Cuenta: 4, a: 5, b: 10
Cuenta: 5, a: 5, b: 10
```

Este ciclo **while** de Python tiene múltiples condiciones que deben ser evaluadas conjuntamente. La primera condición comprueba si cuenta es `menor` que `a`. La segunda condición comprueba si `cuenta` es menor que `b`. La tercera condición utiliza el operador lógico **not** para comprobar que el valor de `cuenta` no ha alcanzado el número máximo de iteraciones. Hemos fijado este valor en 5. Por lo tanto, el ciclo se ejecuta durante cinco iteraciones, y luego termina.

Python evalúa las expresiones condicionales de izquierda a derecha, comparando dos a la vez. Se pueden utilizar paréntesis para visualizar mejor cómo funciona esto:

```
1 while ( EXPRESIÓN CONDICIONAL A or EXPRESIÓN CONDICIONAL B ) and not
EXPRESIÓN CONDICIONAL C
```

Python agrupa las dos primeras expresiones condicionales, y las evalúa juntas. En el bloque de código anterior, `cuenta` se compara con `a` y luego con `b`. El operador lógico **or** que combina estas dos expresiones dice que devuelve *true* si cualquiera de ellas es verdadero. Este es el caso de las cinco iteraciones del bucle (cuenta es siempre menor que `a` o `b`).

A continuación, Python combina el resultado de la primera evaluación con la tercera condición:

```
1 while ( RESULTADO ANTERIOR and not EXPRESIÓN CONDICIONAL C )
```

Aquí, el RESULTADO ANTERIOR será verdadero o falso. Python utiliza el operador lógico, y para combinar esto con la tercera condición. Si ambas son verdaderas, entonces el cuerpo del bucle continuará ejecutándose. Si una es falsa, entonces el bucle terminará.

Observa que el bucle termina en la sexta iteración. Esto es algo que no se observa en la salida de la terminal, pues la sexta iteración hace que la sentencia **while** se evalúe como falsa. Aquí, cuenta es todavía `menor` que a o `b`, pero se ha alcanzado el número máximo de iteraciones. Aunque la segunda condición sigue siendo verdadera, requiere que la tercera condición también sea verdadera. Como es falsa, el cuerpo del ciclo termina.

MANEJANDO CONDICIONES DE BORDE

Un tema relevante a la hora de usar ciclos es lo que se denomina condiciones de borde. En resumen, las condiciones de borde son restricciones necesarias para la solución de un problema en donde la solución debe cumplir un requisito. A estos tipos de problema se les llama condiciones de borde. Por ejemplo, ¿qué pasa si una aplicación requiere que un usuario ingrese un número par y, sin embargo, el usuario solo ingresa un número impar? ¿Debería dejar de funcionar la aplicación o debería no hacer nada? No, aquí es donde se deben usar las condiciones de borde para que la aplicación pueda dar retroalimentación al usuario y así cumplir su objetivo. Dependiendo de lo que requieran nuestras aplicaciones, podemos manejar las condiciones de borde necesarias para proporcionar el comportamiento efectivo de nuestro programa al usuario.

CICLOS INFINITOS

Un ciclo infinito en Python es un bucle condicional continuo y repetitivo, el cual se ejecuta hasta que un factor externo interfiere en el flujo de ejecución, como una memoria insuficiente de la CPU, un código de función/error fallido que detuvo la ejecución, o una nueva función en los otros sistemas heredados que necesita la integración del código.

Puede ser útil en la programación cliente/servidor, donde el servidor necesita funcionar con continuidad para que los programas cliente puedan comunicarse con el programa servidor cuando surja la necesidad. También si se necesita crear una nueva conexión. Existe la utilidad de un bucle **while** en una aplicación de juegos, o en una aplicación en la que entramos en una especie de ciclo de evento principal, que continúa ejecutándose hasta que el usuario selecciona una acción para romper ese bucle infinito. También, si uno tiene que jugar un juego, y desea que éste se reinicie después de cada sesión. Las iteraciones son el proceso de hacer una tarea repetitiva, y los programas de ordenador siempre han dominado este arte.

Sin embargo, aunque en muchos casos es de gran utilidad, es posible crear un ciclo infinito por error y, de hecho, es muy común de cometer. La forma más conocida es crear una condición y luego olvidar programar el elemento que se actualizará y evaluará en cada ciclo que deba cumplir esta condición.

Un ejemplo muy común es crear un contador, y luego olvidar actualizarlo.

```
i = 0
while i <= 10:
    print("He creado un ciclo infinito")
```

La razón por la cual hemos creado un ciclo infinito es porque nunca cambiamos el valor de `i`, siempre será cero y por tanto siempre será inferior a 10.

La programación correcta es:

```
i = 0
while i <= 10:
    print("He creado un ciclo infinito")
    i += 1
```

En cuyo caso el ciclo correrá hasta que `i` sea mayor a 10.

UTILIZANDO CICLOS ANIDADOS

Si un ciclo existe dentro del cuerpo de otro ciclo, se denomina Ciclo Anidado. Esto significa que queremos ejecutar el código del bucle interno varias veces. El bucle externo controla el número de iteraciones que sufrirá el bucle interno.

La sintaxis básica de un ciclo **for** anidado en Python es:

```
#Bucle exterior
for variable1 in secuencia1:
    #Bucle interior
    for variable2 in secuencia2:
        código a ejecutar
```

Un ejemplo:

```
1  for i in range(5):
2      for j in range(i):
3          print('*', end="")
4      print("")
5
6  #Resultado
7  *
8  **
9  ***
10 ****
11 ****
```

Intentemos entender el flujo de ejecución del programa anterior. En el programa, utilizamos dos variables de iteración, `i` y `j`, para imprimir un patrón de estrellas.

El compilador comienza con la línea 1. Se encuentra con un ciclo **for,** y una función `range`. La función `range` de Python produce una lista iterable de números enteros desde 0 hasta el número especificado en el argumento. El número del argumento se excluye de la matriz. En nuestro caso, generará una matriz [0, 1, 2, 3, 4]. Ahora, el compilador sabe que debe ejecutar el siguiente conjunto de sentencias 4 veces.

Cuando pasa a la línea 2, se encuentra con otro ciclo **for** y una función de rango. Obsérvese que el argumento de esta función `range` es un valor calculado de nuestra variable de iteración `i`. Así, genera dinámicamente una lista dependiendo del valor de `i`. Cuando `i=0`, la lista está vacía. Cuando `i=1`, la lista es `[0]`. Cuando `i=2`, la lista es `[0, 1]`, y así sucesivamente.

Así, el número de veces que se ejecuta la línea 3 depende directamente del valor de `i`. Fíjate en la parte `end=''` en la línea 3. Esto es para evitar que Python imprima un salto de línea después de cada estrella. Sólo queremos un salto de línea al final de cada iteración del bucle exterior. Hemos impreso explícitamente un salto de línea en la línea 4 de nuestro código.

La sintaxis para anidar el ciclo **while** en Python es:

```
#Bucle exterior
while expresion1:
    #Opcional
    Código a ejecutar
    #Bucle interior
        while (expresión_2):
            Código a ejecutar
```

Un ejemplo que podemos observar es:

```
i=1
while(i<=5):
    j=5
    while(j>=i):
        print(j, end=' ')
        j-=1
    i+=1
    print()

#Resultado:
5 4 3 2 1
5 4 3 2
5 4 3
5 4
5
```

La línea 1 del código establece la variable de iteración del ciclo exterior al valor inicial. La siguiente línea es el comienzo del bucle exterior **while**. Tiene una expresión `i <=5`. Esta expresión se evalúa para obtener un valor verdadero después de cada iteración. La ejecución entra en el ciclo sólo si la condición es verdadera. Tan pronto como la condición se convierte en falsa, el ciclo se termina.

Como el valor inicial de `i` es 1, la condición en la línea 2 es verdadera. Por lo tanto, el compilador se mueve a la línea 3, y establece la variable iteradora `j` de nuestro ciclo interno a 5. La línea 4 tiene de nuevo un ciclo **while** con una expresión que se evalúa como verdadera. Así, el compilador ejecuta las líneas 5 y 6. Luego vuelve a la línea 4, y evalúa la condición. Si la condición es verdadera, vuelve

a entrar en la línea 5 y 6. Si la condición es falsa, el ciclo se termina, y las siguientes líneas a ejecutar son la 7 y la 8. Lo mismo se sigue para el bucle exterior.

El bucle **while** sigue ejecutando el código hasta que la expresión se evalúa como falsa. Por lo tanto, un desarrollador debe tener siempre presente la actualización de la variable/expresión iterable, o de lo contrario el bucle entrará en modo de ejecución infinita.

COMBINACIÓN DE CICLOS CON INSTRUCCIONES IF/ELSE

Dentro de un ciclo for y/o **while**, también se pueden utilizar sentencias **if**, **elif** y **else**. Cada ciclo puede tener una serie de acciones condicionadas que pueden ser determinadas con las sentencias **if**.

Por ejemplo: una sentencia **if** dentro de un ciclo **for** es perfecta para evaluar una lista de números en un rango (o elementos en una lista) y ponerlos en diferentes cubos, etiquetarlos, o aplicarles funciones - o simplemente imprimirlos.

Veamos un ejemplo con el siguiente enunciado: “Recorrer todos los números hasta el 99. Imprime "fizz" para cada número divisible por 3, imprime "buzz" para cada número divisible por 5, e imprime "fizzbuzz" para cada número divisible por 3 y por 5. Si el número no es divisible ni por 3 ni por 5, imprime un guion ('-')”.

```
1 for i in range(100):
2     if i % 3 == 0 and i % 5 == 0:
3         print('fizzbuzz')
4     elif i % 3 == 0:
5         print('fizz')
6     elif i % 5 == 0:
7         print('buzz')
8     else:
9         print('-')
```

Recuerda que cuando utilices una sentencia **if** dentro de un ciclo **for**, debes tener cuidado con la indentación, pues si las colocas mal se producen errores o resultados falsos.

UTILIZANDO LAS SENTENCIAS BREAK Y CONTINUE

La sentencia **break** de Python detiene el ciclo en el que se encuentra la sentencia. Cuando se ejecuta una sentencia **break**, se ejecutan las sentencias posteriores al contenido del ciclo.

Una sentencia **break** puede colocarse dentro de un bucle anidado. Si una sentencia **break** aparece en un bucle anidado, sólo el ciclo interior dejará de ejecutarse. El ciclo exterior continuará ejecutándose hasta que se hayan producido todas las iteraciones, o hasta que el ciclo exterior se rompa utilizando una sentencia **break**.

Se pueden utilizar sentencias **break** para salir de un ciclo cuando se cumple una condición específica. Se declara una sentencia **break** dentro del ciclo, normalmente bajo una sentencia **if**.

Por ejemplo, es posible que tengamos una lista de nombres de estudiantes para imprimir. Queremos que el programa se detenga después de imprimir el segundo nombre. Esto nos permitirá verificar que éste funciona. Aquí hay un ejemplo de un programa que utiliza una sentencia **break** para hacerlo:

```
estudiantes = ["Paul", "Luis", "Carmen", "Alicia"]

for estudiante in range(0, len(estudiantes)):
    if estudiante == 2:
        break
    else:
        print(estudiantes [estudiante])
              print("Contador es " + str(estudiante))
#Resultado:
Paul
Contador es 0
Luis
Contador es 1
```

El programa hizo la iteración tres veces, en la tercera la variable estudiante adquirió el valor 2, y activó el **break**.

La sentencia **continue** ordena a un ciclo continuar con la siguiente iteración. Cualquier código que siga a la sentencia **continue** no se ejecuta. A diferencia de una sentencia **break**, una **continue** no detiene completamente un ciclo.

Puedes usar una sentencia **continue** en Python para saltarte parte de un ciclo cuando se cumple una condición. Entonces, el resto del ciclo continuará ejecutándose. Las sentencias **continue** se utilizan dentro de los ciclos, normalmente después de una sentencia **if**.

Utilicemos un ejemplo para ilustrar cómo funciona la sentencia **continue** en Python. En el siguiente ejemplo, utilizamos una sentencia **continue** para omitir la impresión del segundo nombre de nuestra lista y continuar iterando:

```
estudiantes = ["Paul", "Luis", "Carmen", "Alicia"]

for estudiante in range(0, len(estudiantes)):
    if estudiante == 2:
        continue
    else:
        print(estudiantes [estudiante])
        print("Contador es " + str(estudiante))

#Resultado:
Paul
Contador es 0
Luis
Contador es 1
Alicia
Contador es 3
```

Nuestra sentencia **continue** se ejecuta cuando se dispara la condición estudiante es igual a 2, el programa deja de ejecutar esa iteración del ciclo, y continúa en la siguiente.