2.4.- Contenido 4: Codificar un programa en Python utilizando sentencias iterativas para resolver problemas cíclicos y condicionales

Objetivo de la jornada

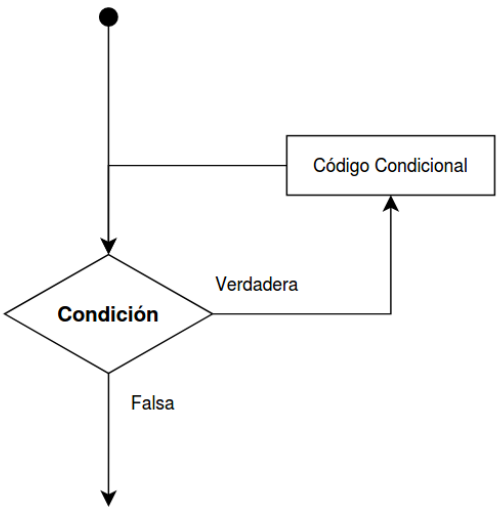
1. Codifica un programa en Python utilizando diagramas de flujo con iteraciones para la implementación de un algoritmo que resuelve un problema.
2. Codifica un programa en Python utilizando ciclos anidados y condiciones de salida para resolver un problema.
3. Codifica un programa en Python utilizando ciclos de instrucciones iterativas combinadas con sentencias if/else para resolver un problema.

2.4.1.- Control de flujo en Python.

2.4.1.1. Para qué sirven los ciclos.

LOOPS (Ciclos)

Desde los primeros años de la historia de la programación surgió la necesidad de automatizar tareas que requerían la ejecución de un mismo código multiples veces. Esto con el fin de realizar la misma tarea una y otra vez, o para iterar aplicando distintos valores de una variable presente en ese bloque de código. Las herramientas que proveen tales facilidades son las estructuras cíclicas, de iteración o de loop (bucle). El siguiente diagrama ilustra, en general, una declaración de bucle como las que mencionamos:



El lenguaje de programación Python proporciona los siguientes tipos de bucles para implementar soluciones a necesidades iterativas:

While
Repite una sentencia o grupo de sentencias mientras una condición determinada es VERDADERA. Prueba la condición antes de ejecutar el cuerpo del bucle.

For
Ejecuta una o múltiples sentencias varias veces y abrevia el código que meneja la variable de loop.

SENTENCIAS DE CONTROL
Las sentencias de control de loop cambian la ejecución de su secuencia normal. Cuando la ejecución abandona un alcance (Scope), todos los automáticos que se crearon en ese scope se destruyen.

Python admite las siguientes sentencias de control:

Break
Interrumpe el loop y transfiere la ejecución a la instrucción que le sigue inmediatamente.

Continue
Omita el resto del bloque de código de la presente iteración del loop e inmediatamente pasa a la siguiente iteración.

Pass
Se usa cuando no se requiere ejecutar ningún comando o código, pero sí tener estructurado el programa en un formato de loop, tal vez para agregar código posteriormente.

A continuación revisaremos en mayor detalle los loops **while** y **for**, así como también las sentencias de control **break**, **continue** y **pass**.

2.4.1.2. El Loop While

La sentencia de loop while en el lenguaje de programación Python ejecuta repetidamente un bloque de código de destino siempre que una condición determinada sea verdadera.

SINTAXIS
La sintaxis de un loop while en el lenguaje de programación Python es:

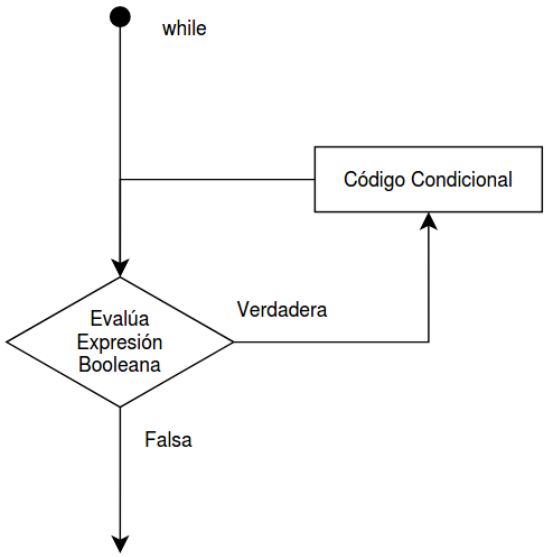
```
while <expresión booleana>:
   Bloque de código
```

Aquí, “bloque de código” pueden ser una sola sentencia o un conjunto de éstas. La condición puede ser cualquier expresión booleana y verdadero es cualquier valor distinto de cero. El loop se repite mientras la condición es verdadera.

Cuando la condición se vuelve falsa, el control del programa pasa a la línea inmediatamente siguiente al loop.

En Python, todas las sentencias indentadas por el mismo número de espacios (4) de caracteres se consideran parte de un solo bloque de código, en este caso para estar dentro de la estructura las líneas deberán estar identadas 4 espacios respecto de la palabra **while**.

Si queremos representar la estructura While como un diagrama de flujo podemos ilustralo de la siguiente forma:



Como puede verse el ciclo while es posible que éste nunca se ejecute. Cuando se evalúa la expresión booleana y el resultado es falso, el control de ejecución del programa saltará a la primera instrucción posterior al loop **while**.

Ejemplo

```
contador = 0
while (contador < 9):
   print(“El contador es:”, contador)
   contador = contador + 1
print("Fin!")
```

Cuando este código es ejecutado, se entrega el siguiente resultado:

```
El contador es: 0
El contador es: 1
El contador es: 2
El contador es: 3
El contador es: 4
El contador es: 5
El contador es: 6
El contador es: 7
El contador es: 8
Fin!
```

El bloque de código condicional consta de una sentencia de impresión y otra para incremento del contador. Esto se ejecuta repetidamente hasta que el contador no es menor que 9. Con cada iteración se muestra el valor actualizado del contador y luego éste se incrementa en 1.

LOOP INFINITO

Un loop se convierte en infinito si su condición nunca se vuelve FALSA. Se debe ser cuidadoso al usar loops **while** debido a la posibilidad de que esta condición nunca se resuelva en un valor FALSO. Esto da como resultado un ciclo que nunca termina.

Un loop infinito puede ser útil, por ejemplo, en la programación cliente / servidor donde el servidor necesita ejecutarse continuamente para que los programas cliente puedan comunicarse con él cuando sea necesario.

Por ejemplo, el siguiente código atiende permanentemente en un loop infinito al usuario para sumar dos números.

```
while True:
    print("SUMADOR DE DOS ENTEROS:")
    numero_1 = int(input("Primer entero: "))
    numero_2 = int(input("Segundo entero: "))
    suma = numero_1 + numero_2
    print(suma)
```

Al ejecutar este programa vemos que las instrucciones para solicitar al usuario ingresar los sumandos y efectuar la operación se repiten sin fin:

```
$ python sumador.py
SUMADOR DE DOS ENTEROS:
Primer entero: 5
Segundo entero: 4
9
SUMADOR DE DOS ENTEROS:
Primer entero: 7
Segundo entero: 5
12
SUMADOR DE DOS ENTEROS:
Primer entero: 3
Segundo entero: 9
```

```
12
SUMADOR DE DOS ENTEROS:
Primer entero: 2
Segundo entero: 3
5
```

Dada la ejecución en loop infinito, la forma de detener la ejecución es a través de el uso de **Ctrl + C**, con lo que recuperamos el control del programa.

USO DE WHILE CON INSTRUCCIÓN ELSE

Python admite tener una instrucción **else** asociada con una estructura de loop. Si la instrucción **else** se usa con un loop **while**, la instrucción **else** se ejecuta cuando la condición del loop se vuelve falsa.

El siguiente ejemplo (Lado izquierdo) ilustra la combinación de una instrucción **else** con una instrucción **while** que imprime un número siempre que sea menor que 5; de lo contrario, se ejecuta la instrucción **else**.

while_else.py

```
cuenta = 0
While cuenta < 5:
   print(cuenta, " es menor a 5")
   cuenta = cuenta + 1
else:
   print(cuenta, " no es menor a 5")
```

while_else.py

```
cuenta = 0
while cuenta < 5:
   print(cuenta, " es menor a 5")
   cuenta = cuenta + 1
print(cuenta, " no es menor a 5")
```

Podríamos pensar que este código es idéntico al mostrado al costado derecho. Ambos generan el mismo resultado:

```
$ python while_else.py
0  es menor a 5
1  es menor a 5
2  es menor a 5
3  es menor a 5
4  es menor a 5
5  no es menor a 5
```

A pesar de entregar el mismo resultado, el programa del lado izquierdo sólo imprime "5 no es menor a 5" cuando se vuelve falsa la condición de la sentencia **while**. Sin embargo, el de la derecha imprime "5 no es menor a 5" incluso en casos en que el loop se interrumpe por otros motivos, como por una instrucción **break** (Que estudiaremos más abajo). El código de la izquierda en caso de un **break** no ejecuta el código dentro de **else**:

while_else.py

```
cuenta = 0
while cuenta < 5:
    print(cuenta, " es menor a 5")
    cuenta = cuenta + 1
```

while_else.py

```
cuenta = 0
while cuenta < 5:
    print(cuenta, " es menor a 5")
    cuenta = cuenta + 1
```

```
    if cuenta == 4:
        break
else:
    print(cuenta, " no es menor a 5")
```

```
    if cuenta == 4:
        break
print(cuenta, " no es menor a 5")
```

En este caso cuando la variable **cuenta** es igual a 4 se ejecuta la instrucción **break** que interrumpe el loop y el mensaje " no es menor a 5" no se despliega, a diferencia de la salida entregada por el código de la derecha. En este caso el resultado es:

```
$ python while_else.py
0  es menor a 5
1  es menor a 5
2  es menor a 5
3  es menor a 5
```

```
$ python while_else.py
0  es menor a 5
1  es menor a 5
2  es menor a 5
3  es menor a 5
4  no es menor a 5
```

WHILE DE UNA SOLA LÍNEA

Al igual que en la sintaxis de la instrucción **if**, si un while consta de una única instrucción, puede colocarse en la misma línea que el encabezado de **while**. De esta forma podríamos escribir lo siguiente:

```
while True: print('Loop while infinito en una línea')
```

que entrega una repetición infinita de la línea “Loop while infinito en una línea”.

2.4.1.3. El Loop For

El loop **for** es la estructura condicional más preferida para ser utilizada en un programa Python. Es la más usada cuando es conocido el número total de iteraciones necesarias para la ejecución.

SINTAXIS

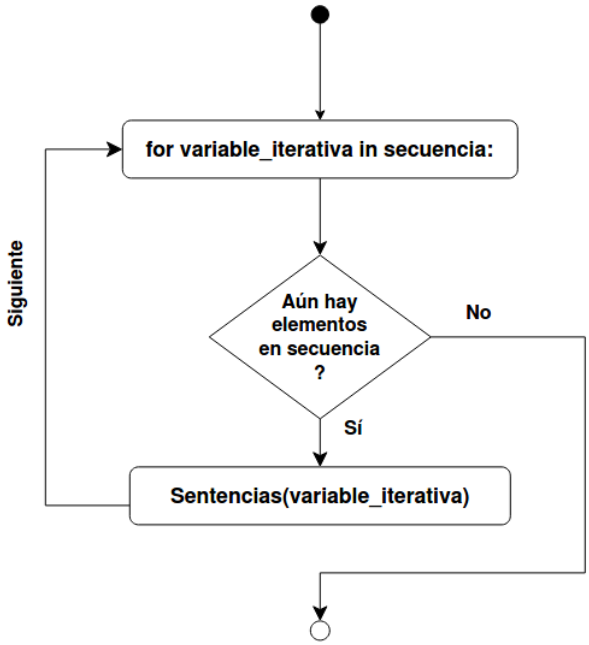
Tiene una sintaxis más clara y simple, y se puede utilizar para recorrer diferentes tipos de secuencias. Python admite iterar sobre tipos de datos tales como: Strings, Listas, Tuplas, Bytearray y objetos Range. También hay conjuntos y diccionarios, pero son solo contenedores para los tipos de secuencia.

Se puede usar el loop **for** de la siguiente manera:

```
for variable_iterativa in secuencia:
    sentencias(variable_iterativa)
```

variable_iterativa representa una variable, a la que se le asigna sucesivamente un valor obtenido desde el objeto **secuencia**.

secuencia puede hacer referencia a objetos de Python, tales como una Lista, una Tupla, un String o un Range.



La lógica de este diagrama es que si **secuencia** contiene una lista de expresiones, primero se evalúa. Luego, el primer elemento de la secuencia se asigna a **variable_iterativa**. A continuación, se ejecuta el bloque de sentencias. Cada elemento de la lista se asigna a variable_iterterativa en forma sucesiva y el bloque de sentencias se ejecuta, hasta que se recorren todos los elementos de la secuencia.

LOOS FOR CON SECUENCIA DADA POR RANGE()

La forma más básica de utilizar un loop **for** es para repetición de un número determinado de veces de una sentencia, tal como se muestra en este ejemplo:

```
for i in range(3):
    print('Mensaje repetitivo')
```

En este caso la variable iterativa **i** no es utilizada dentro del bloque de código interno del loop, y se usa sólo como variable muda. En este caso la salida es:

```
Mensaje repetitivo
Mensaje repetitivo
Mensaje repetitivo
```

También podemos hacer uso de la variable iterativa i, por ejemplo imprimiéndola siguiendo el texto del mensaje reperititivo:

```
for i in range(3):
    print('Mensaje repetitivo', i)
```

Lo que se traduce en:

```
Mensaje repetitivo 0
Mensaje repetitivo 1
Mensaje repetitivo 2
```

La función range() puede ser utilizada de manera más sofisticada. Para eso debemos considerar su sintaxis general:

```
range(inicio,parada,paso)
```

Cuando más arriba llamamos a **range(3)**, se obtiene la misma salida que si llamáramos a **range(0,3,1)** (**inicio = 0, parada = 3, paso = 1**). Debemos notar que el conteo en Python comienza en **0** y termina en **n-1**. Si solo se proporcionan dos argumentos, como en **range(0,3)**, se supone un **paso = 1**.

La siguiente tabla muestra ejemplos de la función **range()** de Python y la salida asociada:

Función range()	Salida
range(3)	0, 1, 2
range(0,3)	0, 1, 2
range(0,3,1)	0, 1, 2
range(2,7,2)	2, 4, 6
range(0,-5,-1)	0, -1, -2, -3, -4
range(2,-3,1)	(sin salida)

LOOP FOR CON SECUENCIA DADA POR UNA LISTA

El objeto **secuencia** de la estructura iterativa **for** puede estar data por una lista Python. En este caso en cada una de las iteraciones, los elementos de la lista serán pasados secuencial y ordenadamente a **variable_iterativa**, la que podrá ser utilizada en las sentencias al interior del loop.

Por ejemplo, consideremos una lista que contenga un listado de árboles e iteremos sobre ella con un loop for:

for_list.py

```
arboles = ["peumo", "maitén", "calafate", "lenga", "ulmo"]
for arbol in arboles:
    print(arbol)
```

Como podemos ver en este caso no hay una variable iterativa numérica explícita que se vaya incrementando en cada iteración. En su lugar, for tiene la habilidad de manejar toda la mecánica de manejo de indices para iterar sobre cada uno de los elementos de la lista **arboles**. Esto se traduce en una manera muy intuitiva de utilizar

la estructura **for** con una analogía a la frase “Para cada **arbol** en la lista de **arboles**”. De esta forma, cada nombre de árbol es asignado a la variable **arbol** y es utilizada, en este caso para imprimir su valor. La salida de este programa por lo tanto es:

```
$ python for_list.py
peumo
maitén
calafate
lenga
Ulmo
```

Ejemplo loop for y listas

A continuación ilustraremos la utilizacion del loop for en un programa con un mayor grado de sofisticación. Usaremos los siguientes pasos para calcular la suma de N números. Supongamos que nos entregan los siguientes requerimientos:

1. Cree una lista de números enteros y complétela con **N (= 6)** valores.
2. Inicialice una variable (**suma = 0**) para almacenar la suma.
3. Itere **N (= 6)** número de veces para obtener el valor de cada número entero de la lista.
4. En el loop, sume el entero de la iteración en curso al valor actual de la variable **suma** y actualice esta última con el nuevo valor.
5. Divida la suma fonal pòr **N (= 6)**. Usamos la función **len()** para determinar el tamaño de nuestra lista. De esta forma habremos obtenido el promedio.
6. Finalmente, imprima tanto la suma como el promedio.

A continuación se muestra el código Python para el programa anterior:

for_list_2.py

```
lista_enteros = [1, 2, 3, 4, 5, 6]
suma = 0
for entero in lista_enteros:
    suma = suma + entero
print("Suma =", suma)
print("Promedio =", suma/len(lista_enteros))
```

Lo que resulta en la siguiente salida:

```
$ python for_list_2.py
Suma = 21
Promedio = 3.5
```

LOOP FOR CON SECUENCIA DADA POR UN STRING

Los loops **for** también pueden utilizarse con el objeto **secuencia** dado por un string y **variable_iterativa** será secuencialmente cada uno de los caracteres de éste. Un loop definido por un string se ejecuta tantas veces como caracteres haya en éste.

Un ejemplo de un loop de este tipo es el siguiente, en el que recorremos el string carácter por carácter para utilizarlo dentro del código interno del loop:

for_string.py

```
for letra in "Inteligencia Artificial":
    print("Extracción de la letra: ", letra)
```

Que nos entrega:

```
$ python for_string.py
Extracción de la letra:  I
Extracción de la letra:  n
Extracción de la letra:  t
Extracción de la letra:  e
Extracción de la letra:  l
Extracción de la letra:  i
Extracción de la letra:  g
Extracción de la letra:  e
Extracción de la letra:  n
Extracción de la letra:  c
Extracción de la letra:  i
Extracción de la letra:  a
Extracción de la letra:
Extracción de la letra:  A
Extracción de la letra:  r
Extracción de la letra:  t
Extracción de la letra:  i
Extracción de la letra:  f
Extracción de la letra:  i
Extracción de la letra:  c
Extracción de la letra:  i
Extracción de la letra:  a
Extracción de la letra:  l
```

LOOP FOR CON SECUENCIA DADA POR UNA TUPLA

Muy similar a lo que explicamos para una lista, la estructura for en Python tmabién puede recorrer una tupla para utilizar cada uno de sus elementos en la ejecución delo bloque de código interno. En este caso **variable_iterativa** es cargada secuencialmente con cada uno de los elementos de la tupla que pasa a ser el objeto **secuencia**. De esta forma, el siguiente ejemplo:

for_tuple.py

```
tupla = (10,20,30,40,50)
for elemento in tupla:
    print(elemento)
```

Nos entrega el siguiente resultado:

```
$ python for_tuple.py
10
20
30
```

```
40
50
```

LOOP FOR CON SECUENCIA DADA POR UN DICCIONARIO

Hay dos formas de iterar a través de un objeto de diccionario de Python. Una es buscar el valor asociado para cada clave en la lista **keys()**.

for_dictionary.py

```
diccionario = {'1':'a', '2':'b', '3':'c'}
for clave in diccionario.keys():
    print (clave, diccionario[clave])
```

Y obtenemos el siguiente resultado:

```
$ python for_dictionary.py
1 a
2 b
3 c
```

También, en Python los diccionarios poseen un método llamado **items()** que entrega una lista de tuplas, cada tupla tiene una clave y un valor. Luego, cada tupla se descompone en dos elementos que corresponden a clave y valor de ese elemento del formato diccionario.

for_dictionary_2.py

```
diccionario = {'1':'a', '2':'b', '3':'c'}
for clave, valor in diccionario.items():
    print (clave, valor)
```

Lo que nos da como resultado:

```
$ python for_dictionary_2.py
1 a
2 b
3 c
```

LOOP FOR CON USO DE INSTRUCCIÓN ELSE

Python admite tener una declaración **else** asociada con un loop **for**, tal como lo vimos para el caso de **while**.

Si la instrucción **else** se usa con un loop **for**, la instrucción **else** se ejecuta cuando el loop ha agotado los elementos de la **secuencia**.

El siguiente ejemplo ilustra la combinación de una instrucción else con una instrucción **for** que busca números primos entre los números 10 y 20.

for_else.py

```
# Números primos entre dos números
```

```
inicial = 1
final = 10
print("Números primos entre", inicial, "y", final, ":")
for num in range(inicial, final + 1):
    # all prime numbers are greater than 1
    if num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                j=num/i
                print('%d no es primo: %d * %d = %d' % (num,i,j, num))
                break
        else:
            print(num, "es primo!")
```

En este caso el loop **for** más interno itera sobre desde el entero **2** hasta el **num-1**. Para cada uno de ellos verifica si puede dividir de manera justa al número **num**. Si esto no ocurre se cumple el loop **for** completo y pasa a ejecutarse la sentencia **print()** que indica que el número es primo. Si existe un entero que divida en forma exacta al número **num**. Entonces, se indica que **num** no es un número primo y se ejecuta la instrucción **break**, que termina en forma anticipada el loop **for** más interno y da paso a la siguiente iteración de la variable **num**. Así, el resultado de la ejecución del código es:

```
$ python for_else.py
Números primos entre 1 y 10 :
2 es primo!
3 es primo!
4 no es primo: 2 * 2 = 4
5 es primo!
6 no es primo: 2 * 3 = 6
7 es primo!
8 no es primo: 2 * 4 = 8
9 no es primo: 3 * 3 = 9
10 no es primo: 2 * 5 = 10
```

Se sugiere al lector implementar, como ejercicio, modificar este último código para realizar los mismos cálculos pero sin utilizar la instrucción **else**. Esto le permitirá comprender más claramente la funcionalidad de la instrucción **else** dentro de un loop **for**.

2.4.1.4. Sentencias de Control de Loop (break, continue y pass)

Las sentencias de control de loop cambian la ejecución de la secuencia normal de éste. Cuando la ejecución del programa sale de un alcance (Scope), todos los objetos automáticos que se crearon en ese ámbito se destruyen.

Python admite las siguientes sentencias de contro: **break**, **continue** y **pass**.

BREAK

La sentencia **break** se utiliza para la terminación prematura del loop actual. Después de abandonar el loop, se reanuda la ejecución en la siguiente instrucción posterior al loop, al igual que la instrucción break tradicional en C.

El uso más común de **break** es cuando se presenta una condición externa que requiere una salida apresurada desde el loop. La sentencia **break** se puede utilizar tanto en loops **while** como **for**.

En caso de loops anidados (Tratados más adelante), la instrucción **break** detiene la ejecución del loop más interno y comienza a ejecutar la siguiente línea del más externo.

Sintaxis y Flujo
La sintaxis es simple y sólo consiste en ubicar **break** en la posición del código interno de un loop, donde éste requiera terminarse anticipadamente.

A continuación revisamos un ejemplo que implementa dos loops, en los que se incluye una condición en la cual instruimos al programa a interrumpir el flujo normal del loop correspondiente.

break.py
```
# primer ejemplo de loop con break
for caracter in 'Mesopotamia':
    if caracter == 't':
        break
    print ('Caracter actual :', caracter)
# segundo ejemplo de loop con break
contador = 10
while contador > 0:
    contador = contador - 1
    if contador == 5:
        break
    print ('Valor actual del contador :', contador)
print("Hasta pronto!")
```

El loop for recorre el string 'Mesopotamia' caracter por caracter y los imprime cada uno en una línea nueva. Al llegar al caracter 't' el programa está instruido para detener el flujo normal del loop **for** con la instrucción **break**. De esta forma el control del programa pasa a la siguiente línea posterior al loop **for.** De esta forma para a ejecutarse el segundo ejemplo, donde un loop while itera sobre una función **print()** que imprime el valor de un contador regresivo. Normalmente el valor del contador debería continuar decreciendo hasta llegar a 1. Sin embargo, se ha establecido, a través de un condicional **if**, que cuando el contador tome el valor 5 el loop while sea interrumpido y el control del programa pase a la siguiente instrucción posterior al loop. De esta forma se imprime el mensaje "Hasta pronto!".

```
$ python break.py
Caracter actual : M
Caracter actual : e
Caracter actual : s
Caracter actual : o
```

```
Caracter actual : p
Caracter actual : o
Valor actual del contador : 9
Valor actual del contador : 8
Valor actual del contador : 7
Valor actual del contador : 6
Hasta pronto!
```

https://www.tutorialspoint.com/python3/python_break_statement.htm

CONTINUE

La instrucción **continue** en Python devuelve el control del programa al comienzo del loop pero en la siguiente iteración. Las sentencias siguientes a la instrucción **continue** , que están dentro del loop, no se ejecutarán para la iteración actual. Como puede apreciarse, el comportamiento de **continue** es similar al de **break**, con la diferencia que en lugar de salir totalmente del loop **continue** sólo da inicio anticipado a la siguiente iteración.
La instrucción continue se puede utilizar tanto en loops **while** como **for**.

Sintaxis y flujo

La sintaxis es simple y sólo consiste en ubicar **continue** en la posición del código interno de un loop, donde éste requiera interrumpir y dar inicio anticipadamente a la siguiente iteración de éste.

A continuación revisamos el mismo ejemplo que analizamos para la instrucción **break**, pero reemplazándola por **continue**.

```
# primer ejemplo de loop con continue
for caracter in 'Mesopotamia':
    if caracter == 't':
        continue
    print ('Caracter actual :', caracter)
# segundo ejemplo de loop con continue
contador = 10
while contador > 0:
    contador = contador - 1
    if contador == 5:
        continue
    print ('Valor actual del contador :', contador)
print("Hasta pronto!")
```

Vemos que a diferencia de **break**, la instrucción **continue** no da término al loop sino que sólo deja de ejecutar las siguientes instrucciones siguientes de esa iteración. De esta forma, como se muestra a continuación, se deja de imprimir **'t'** en el loop **for,** y en el loop **while** no se imprime el número **5**:

```
$ python continue.py
Caracter actual : M
Caracter actual : e
Caracter actual : s
```

```
Caracter actual : o
Caracter actual : p
Caracter actual : o
Caracter actual : a
Caracter actual : m
Caracter actual : i
Caracter actual : a
Valor actual del contador : 9
Valor actual del contador : 8
Valor actual del contador : 7
Valor actual del contador : 6
Valor actual del contador : 4
Valor actual del contador : 3
Valor actual del contador : 2
Valor actual del contador : 1
Valor actual del contador : 0
Hasta pronto!
```

PASS

Se usa cuando se requiere escribir una sentencia que no puede quedar vacía, pero no se desea que se ejecute ninguna instrucción.

La instrucción pass es una operación nula; no pasa nada cuando se ejecuta. La instrucción pass también es útil en lugares donde un código eventualmente irá ubicado, pero aún no se ha escrito.

En el ejemplo que dimos para **break** y **continue**, si las reemplazamos por pass, el código entregará la misma salida que si las estructuras condicionales **if** no estuvieran incorporadas. Esto puede ser últil para mantener la estructura de nuestro código, a la espera de instrucciones que incluiremos ahí posteriormente.

El código de ejemplo sería:

```
# primer ejemplo de loop con pass
for caracter in 'Mesopotamia':
    if caracter == 't':
        pass
    print ('Caracter actual :', caracter)
# segundo ejemplo de loop con pass
contador = 10
while contador > 0:
    contador = contador - 1
    if contador == 5:
        pass
    print ('Valor actual del contador :', contador)
print("Hasta pronto!")
```

Y su correspondiente salida:

```
$ python pass.py
Caracter actual : M
Caracter actual : e
Caracter actual : s
```

```
Caracter actual : o
Caracter actual : p
Caracter actual : o
Caracter actual : t
Caracter actual : a
Caracter actual : m
Caracter actual : i
Caracter actual : a
Valor actual del contador : 9
Valor actual del contador : 8
Valor actual del contador : 7
Valor actual del contador : 6
Valor actual del contador : 5
Valor actual del contador : 4
Valor actual del contador : 3
Valor actual del contador : 2
Valor actual del contador : 1
Valor actual del contador : 0
Hasta pronto!
```

2.4.1.5. Loops Anidados

El lenguaje de programación Python permite el uso de un loop dentro de otro. A continuación mostramos algunos ejemplos para ilustrar el concepto.

Sintaxis

```
for variable_iterativa_1 in secuencia_1:
    for variable_iterativa_2 in secuencia_2:
        sentencias(variable_iterativa_2)
    sentencias(variable_iterativa_1)
```

Esto puede ser aplicando tanto a los loops **for** como a los loops **while** que hemos revisado hasta el momento.

La sintaxis para loops anidados **while** es de la siguiente forma:

```
while <expresión booleana 1>:
    while <expresión booleana 2>:
        sentencias
    sentencias
```

Es posible anidar loops de cualquier tipo dentro de otro tipo de loop. Por ejemplo un loop **for** puede estar dentro de un loop **while** y viceversa.

Ejemplo

El siguiente programa utiliza un loop anidado for para mostrar las tablas de multiplicar desde el número 1 hasta el 10.

for_nested.py

```
for i in range(1,11):
```

```
    for j in range(1,11):
        k = i*j
        print (k, end=' ')
    print()
```

La función **print()** del loop más interno posee el argumento **end=' '** que evita saltar a una nueva línea luego de imprimir el contenido en pantalla, que es su comportamiento por defecto, y en su lugar imprime un espacio.
La función print() más externa cumple la función de saltar a una nueva línea.

Salida
Al ejecutar el ejemplo produce el siguiente resultado. La variable **j** iterada por el loop más externo corresponden a cada fila de la estructura matricial entregada como resultado. La variable **i** del loop más interno corresponde al índice que define la columna de cada uno de los elementos. Inicialmente se comienza desde i = 1 (Fila 1) y se comienza a llenar cada elemento de esa fila haciendo cambiar **j** desde 1 hasta 10.

```
$ python for_nested.py

 1  2  3  4  5  6  7  8  9  10
 2  4  6  8 10 12 14 16 18  20
 3  6  9 12 15 18 21 24 27  30
 4  8 12 16 20 24 28 32 36  40
 5 10 15 20 25 30 35 40 45  50
 6 12 18 24 30 36 42 48 54  60
 7 14 21 28 35 42 49 56 63  70
 8 16 24 32 40 48 56 64 72  80
 9 18 27 36 45 54 63 72 81  90
10 20 30 40 50 60 70 80 90 100
```

2.4.2.- Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Matt Harrison, Ilustrated Guide to Python3, 2017.

[3] Eric Matthes, Python Crash Course, 2016.

[4] Python Programming Tutorial.
http://www.trytoprogram.com/python-programming

[5] Python 3 – Loops.
https://www.tutorialspoint.com/python3/python_loops.htm

[6] Problem Solving with Python – Loops.
https://problemsolvingwithpython.com/09-Loops/09.00-Introduction/

[7] Python For Loop – A Complete Guide for Beginners.
https://www.techbeamers.com/python-for-loop/