

El Ciclo For

El Ciclo For	1
¿Qué aprenderás?	2
Introducción	2
¡Comencemos!	2
Ciclo For	3
Iterables	3
La función range	3
Utilizando estructuras de datos en un ciclo For	5
Listas	5
Strings	6
Diccionarios	6
Otras funciones útiles al momento de iterar	7
enumerate()	7
zip()	8
¿Cómo salir de un ciclo for?	8
Ejercicio guiado 1: Búsqueda	9
Ciclos Anidados	11
Ejercicio Guiado 2:	11
Escribiendo las tablas de multiplicar	11



¡Comencemos!

¿Qué aprenderás?

- Reconocer las sentencias de ciclos para la construcción de programas.
- Codificar un programa en Python utilizando ciclos anidados y condiciones de salida para resolver un problema.

Introducción

A diferencia del ciclo while, el ciclo for se caracteriza por ser de tipo finito, en el que de antemano se conoce cuántas veces va a ser necesario iterar.

En este capítulo, veremos su aplicación en Python, ventajas, desventajas y diferencias con respecto al ciclo while.



¡Comencemos!

Ciclo For

La instrucción `for` en Python suele funcionar un poco distinta, ya que normalmente va a iterar en un rango de valores, pero en Python funciona más como un “for each”, es decir, que el `for` iterará en cada elemento de un objeto. Este objeto tiene una característica que lo define: debe ser un iterable.

Su sintaxis es la siguiente:

```
for variable in iterable:
    # se ejecutará código para cada valor del iterable.
    # El código debe estar correctamente indentado.
```

Iterables

Los iterables son estructuras que tienen elementos en los cuales es posible desplazarse desde un elemento a otro en orden, y que corresponden a estructuras de datos, como por ejemplo, listas y diccionarios, etc.

La función range

Un primer iterable y el más común inicialmente, es utilizar la función `range()`, la que permite generar un espacio con un rango de números.

Existen 3 maneras de utilizar `range()`:

```
# Con un sólo valor
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

Cuando se utiliza un solo valor, este corresponde al límite superior. En Python el límite superior se excluye, por lo tanto, el rango comenzará en 0 y terminará en 9 (valor anterior al límite superior).

```
# Con dos valores
for i in range(4,10):
    print(i)
```

```
4
5
6
7
8
9
```

En el caso de utilizar dos valores, estos corresponderán al valor de inicio y al límite superior respectivamente, y al igual que el caso anterior, el límite superior se excluye.

```
# Con tres valores
for i in range(4,10,2):
    print(i)
```

```
4
6
8
```

Al utilizar tres valores, estos corresponderán al valor inicial, al límite superior y al paso (step). El límite superior queda igualmente excluido y el paso indicará cada cuanto es el incremento en la generación del rango.

Además, la función `range()` se verá siempre utilizada dentro de ciclo, ya que al utilizarla por sí sola entrega lo siguiente:

```
print(range(4,10,2))
```

```
range(4,10,2)
```

De hecho, si se chequea el tipo de dato, nos arrojará que es de tipo range, por lo que **inicialmente** se utilizará solo como un generador de rangos en el ciclo `for`.

```
print(type(range(4,10,2)))
```

```
range
```

Utilizando estructuras de datos en un ciclo For

Como se dijo anteriormente cualquier iterable puede ser utilizado dentro de un ciclo **for**.
Acá algunos ejemplos:

Listas

Como primer ejemplo, podemos iterar por todos los elementos de una lista:

```
a = [1,5,8,3,4]
for elemento in a:
    print(elemento)
```

```
1
5
8
3
4
```

Entendamos el código, estamos asignando una lista de números a la variable a. Esta lista la estamos recorriendo con el ciclo for donde queremos que por cada **elemento** dentro del **iterable** a (que es una lista de python) muestre en consola cada elemento.

Strings

Hemos mencionado en otras ocasiones que los Strings son muy similares a las listas, en este caso, los strings también son iterables. Por ejemplo, podemos deletrear una o más palabras:

```
texto = "hola mundo"  
for caracter in texto:  
    print(caracter)
```

```
h  
o  
l  
a  
  
m  
u  
n  
d  
o
```



NOTA: El ciclo `for` permite nombrar la variable iteradora de cualquier manera, por lo tanto, es conveniente utilizar un nombre variable que sea representativo de lo que se está iterando.

Diccionarios

Como dijimos en la unidad anterior, un diccionario se compone de una clave y un valor, es por eso que la manera más común de iterar diccionarios es utilizando `.items()`.

Otra diferencia muy importante, es que en cada iteración se extraerán dos elementos, la clave y el valor:

```
diccionario = {"Nombre": "Carlos",  
               "Apellido": "Santana",  
               "Ocupación": "Guitarrista"}  
  
for clave, valor in diccionario.items():  
    print(f"Mi {clave} es {valor}")
```

```
Mi Nombre es Carlos  
Mi Apellido es Santana  
Mi Ocupación es Guitarrista
```

Otras funciones útiles al momento de iterar

enumerate()

`enumerate()` permite agregar un contador a la iteración, por lo tanto extrae elemento y contador.

Ejemplo:

```
texto = "Esternocleidomastoideo"  
for pos, letra in enumerate(texto):  
    print(f"La letra en la posición {pos} es la {letra}")
```

```
La letra en la posición 0 es la E  
La letra en la posición 1 es la s  
La letra en la posición 2 es la t  
La letra en la posición 3 es la e  
La letra en la posición 4 es la r  
La letra en la posición 5 es la n  
La letra en la posición 6 es la o  
La letra en la posición 7 es la c  
La letra en la posición 8 es la l  
La letra en la posición 9 es la e  
La letra en la posición 10 es la i  
La letra en la posición 11 es la d  
La letra en la posición 12 es la o  
La letra en la posición 13 es la m  
La letra en la posición 14 es la a  
La letra en la posición 15 es la s  
La letra en la posición 16 es la t  
La letra en la posición 17 es la o  
La letra en la posición 18 es la i  
La letra en la posición 19 es la d  
La letra en la posición 20 es la e  
La letra en la posición 21 es la o
```



NOTA: `enumerate()` comienza su conteo en cero.

zip()

Permite unir varios iterables para utilizarlos dentro de la misma iteración:

```
prefijo = ['La', 'El', 'La', 'El']  
frutas = ['manzana', 'platano', 'frutilla', 'tomate']  
colores = ['verde', 'amarillo', 'roja', 'rojo']  
  
for p, fruta, color in zip(prefijo, frutas, colores):  
    print(f'{p} {fruta} es de color {color}')
```

```
La manzana es de color verde  
El platano es de color amarillo  
La frutilla es de color roja  
El tomate es de color rojo
```

¿Cómo salir de un ciclo for?

Previamente, vimos que se puede terminar un ciclo `while` con una condición de salida. En el caso de un ciclo `for`, este termina cuando se recorre todo el iterable, pero ¿es posible terminar un ciclo `for` a propósito?

La respuesta es **sí**, pueden existir varios casos en que se requiera terminar el ciclo sin la necesidad de recorrer todos los elementos. Para hacerlo, se debe escribir la palabra `break`, y esta instrucción hace que el ciclo termine y se continúe con la ejecución del resto del programa.

Ejercicio guiado 1: Búsqueda

Es muy común tener que buscar un elemento específico en la estructura contenedora. Uno podría almacenar el elemento buscado y terminar de recorrer los elementos sin uso de `break`, pero esto no tiene mucho sentido, ya que puede ser muy costoso el recorrer estructuras de datos muy grandes si es que ya se encontró lo que se busca. En este caso buscaremos por un número y diremos en qué posición se encontraba:

1. Creemos el archivo `search.py`
2. Ingreseemos el número a buscar utilizando `sys.argv`

```
import sys
buscar = sys.argv[1] # número a buscar
```

3. Busquemos en una lista de dígitos. Para poner un poco de dificultad podemos mezclar la lista, es importante considerar que acá se muestra un resultado, el cual puede variar según la mezcla que se realice.

```
import random
lista = [1,2,3,4,5,6,7,8,9,0]
# .shuffle de la librería random permite mezclar
# la lista de dígitos para aumentar un poco la dificultad.
random.shuffle(lista)
```

4. Ahora, debemos implementar el algoritmo de búsqueda, el cual podemos conseguir mezclando instrucciones `if` y `for`:

```
# revisaremos cada elemento en la lista
# también llevamos registro de la posición en la que estamos
for position, elemento in enumerate(lista):

# Si el elemento es igual a lo que buscamos terminamos el ciclo
    if elemento == buscar:
        print("¡Elemento encontrado! Se terminará del ciclo")
        break
    else:
# Si es que no es el elemento buscado lo reportamos
        print("Elemento no encontrado")
```

5. Finalmente, reportamos nuestros resultados:

```
print("Ha terminado el ciclo")
print(f'El elemento {buscar} se encontró en la posición {position}')
print(f'La lista mezclada es: {lista}')
```

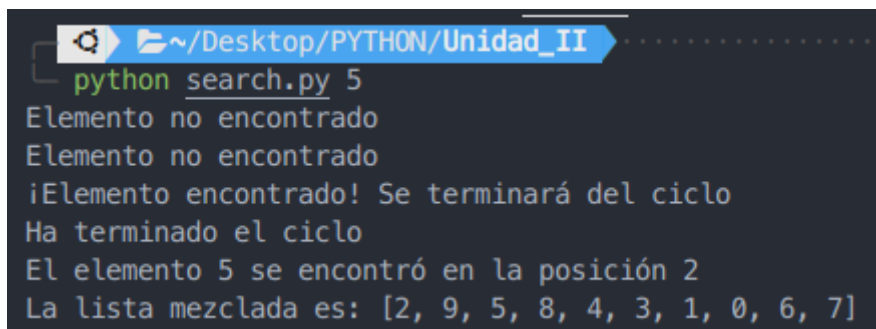
El código final se ve así:

```
1  import sys
2  import random
3
4  buscar = int(sys.argv[1])
5
6  lista = [1,2,3,4,5,6,7,8,9,0];
7  random.shuffle(lista)
8
9
10 for position, elemento in enumerate(lista):
11     if elemento == buscar:
12         print("¡Elemento encontrado! Se terminará del ciclo")
13         break
14     else:
15         print("Elemento no encontrado")
16
17 print("Ha terminado el ciclo")
18 print(f'El elemento {buscar} se encontró en la posición {position}')
19 print(f'La lista mezclada es: {lista}')
```

Imagen 1: Vista al Script completo

Fuente: Desafío Latam

Ejecutando el código se obtiene lo siguiente:



```
python search.py 5
Elemento no encontrado
Elemento no encontrado
¡Elemento encontrado! Se terminará del ciclo
Ha terminado el ciclo
El elemento 5 se encontró en la posición 2
La lista mezclada es: [2, 9, 5, 8, 4, 3, 1, 0, 6, 7]
```

Imagen 2: Ejecutando el programa desde la terminal

Fuente: Desafío Latam

Se puede ver que efectivamente al buscar el número 5 (el argumento provisto a `argv`), se encuentra en la posición 2 (recordar que parte en 0).

En la plataforma tendrás el código de este ejercicio con el nombre **Ejecución guiada - Búsqueda**.

Ciclos Anidados

Un ciclo anidado no es más que un ciclo dentro de otro ciclo, y no existe un límite explícito sobre cuántos ciclos pueden haber anidados dentro de un código, aunque por cada uno aumentará la complejidad.

Se debe tener en cuenta que la cantidad total de iteraciones será la multiplicación de cuántas iteraciones se haga en cada ciclo.



NOTA: Dentro de las buenas prácticas de programación, se recomienda no usar más de 3 ciclos anidados.

Ejercicio Guiado 2:

Escribiendo las tablas de multiplicar

Supongamos que queremos mostrar una tabla de multiplicar, por ejemplo la tabla del 5. Esto se puede escribir como:

```
for numero in range(10):  
    print(f"5 * {numero} = {5 * numero}")
```

```
5 * 0 = 0  
5 * 1 = 5  
5 * 2 = 10  
5 * 3 = 15  
5 * 4 = 20  
5 * 5 = 25  
5 * 6 = 30  
5 * 7 = 35  
5 * 8 = 40  
5 * 9 = 45
```

Ahora bien, ¿cómo podríamos hacer para mostrar todas las tablas de multiplicar del 1 al 10? Fácil, envolviendo el código anterior en otro ciclo que itere de 1 a 10.

El ciclo más externo (`numero1`) será la tabla que nos interesa, mientras que el ciclo externo (`numero2`) serán todos los números que se irán multiplicando con el indicador de la tabla.

```
for numero1 in range(10):  
    print(f'\nTabla del {numero1}:-----\n')  
  
    for numero2 in range(10):  
        print(f"{numero1} * {numero2} = {numero1*numero2}")
```

Tabla del 0:-----

```
0 * 0 = 0  
0 * 1 = 0  
0 * 2 = 0  
0 * 3 = 0  
0 * 4 = 0  
0 * 5 = 0  
0 * 6 = 0  
0 * 7 = 0  
0 * 8 = 0  
0 * 9 = 0
```

Tabla del 1:-----

```
1 * 0 = 0  
1 * 1 = 1  
1 * 2 = 2  
1 * 3 = 3  
1 * 4 = 4  
1 * 5 = 5  
1 * 6 = 6
```

...

```
8 * 6 = 48  
8 * 7 = 56  
8 * 8 = 64  
8 * 9 = 72
```

Tabla del 9:-----

```
9 * 0 = 0  
9 * 1 = 9  
9 * 2 = 18  
9 * 3 = 27
```

```
9 * 4 = 36
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
```

En este caso, se justifica el uso de dos for anidados, porque facilita el problema planteado. Un problema se vuelve complejo si se van anidando ciclos de forma excesiva e innecesaria.

En la plataforma tendrás acceso al código de este ejercicio con el nombre **Ejecución guiada -Escribiendo las tablas de multiplicar**.