INFORME 2 (Informática 3).

CICLO FOR.
Presentado por: Jose David Vazco Loaiza
hora: 2 a 4 pm.

Objetivo general: Conocer las diferentes aplicaciones que tiene el ciclo for en PY. .
Resumen: En este capítulo de la clase veremos la estructura del ciclo for y analizaremos cómo funciona dentro de las diferentes aplicaciones que tiene este bucle o ciclo como lo es llamado en la teoría de py ademàs profundizaremos un poco más en cada elemento que forma su estructura lógica es decir hablaremos más un poco de lo que es un objeto iterable o un iterable y como se puede aplicar de esta manera a tuplas, listas, diccionarios .
Marco teórico: El ciclo for se usa para recorrer los elementos de un objeto iterable (lista, tupla, conjunto, diccionario, …) y ejecutar un bloque de código. En cada paso de la iteración se tiene en cuenta a un único elemento del objeto iterable, sobre el cuál se pueden aplicar una serie de operaciones.
Su estructura es la siguiente:

```
for <x> in <iterable>:
    <Tu código>
```

Aquí, "`x`" es la variable que toma el valor del elemento dentro del iterador en cada paso del bucle. Este finaliza su ejecución cuando se recorren todos los elementos.
Ejemplos de cómo usar el bucle for:

```
nums = [4, 78, 9, 84]
for n in nums:
    print(n)
```

```
4
78
9
84
```

Ahora veremos lo que es un iterable: Un iterable es un objeto que se puede iterar sobre él, es decir, que permite recorrer sus elementos uno a uno. Un iterador es un objeto que define un mecanismo para recorrer los elementos del iterable asociado. Analicemos un ejemplo:

```
>>> nums = [4, 78, 9, 84]
>>> it = iter(nums)
>>> next(it)
4
>>> next(it)
78
>>> next(it)
9
>>> next(it)
84
>>> next(it)
Traceback (most recent call last):
File "<input>", line 1, in <module>
StopIteration
```

Como puedes observar, un interador recorre los elementos de un iterable solo hacia delante. Cada vez que se llama a la función next() se recupera el siguiente valor del iterador. En Python, los tipos principales list, tuple, dict, set o string entre otros, son iterables, por lo que podrán ser usados en el bucle for.

Bucle for en diccionarios:

Un caso es especial de bucle for se da al recorrer los elementos de un diccionario. Dado que un diccionario está compuesto por pares clave/valor, hay distintas formas de iterar sobre ellas.

- recorrer el diccionario:

```
valores = {'A': 4, 'E': 3, 'I': 1, 'O': 0}
for k in valores:
    print(k)

A
E
I
O
```

- Iterar sobre los valores del diccionario:

```
valores = {'A': 4, 'E': 3, 'I': 1, 'O': 0}
for v in valores.values():
    print(v)

4
3
1
0
```

- Iterar sobre la clave y cada uno de los elementos del diccionario:

```
valores = {'A': 4, 'E': 3, 'I': 1, 'O': 0}
for k, v in valores.items():
    print('k=', k, ', v=', v)

k=A, v=4
k=E, v=3
k=I, v=1
k=O, v=0
```

Modificar la estructura del bucle for: Break and continue.

Por último, vamos a ver que es posible alterar la iteración de un bucle for en Python. Para ello, nos valdremos de las sentencias break y continue. Pero, ¿qué hacen estas sentencias?

- break se utiliza para finalizar y salir el bucle, por ejemplo, si se cumple alguna condición.
- Por su parte, continue salta al siguiente paso de la iteración, ignorando todas las sentencias que le siguen y que forman parte del bucle.

Uso del break and continue para encontrar un elemento de una lista:

```
coleccion = [2, 4, 5, 7, 8, 9, 3, 4]
for e in coleccion:
    if e == 7:
        break
    print(e)
```

```
coleccion = [2, 4, 5, 7, 8, 9, 3, 4]
for e in coleccion:
    if e % 2 != 0:
        continue
    print(e)
```

Luego se muestran los números 2,4,8 y 4.

Estructura For else…:

Lo veremos con un ejemplo

```
for e in iterable:
    # Tu código aquí
else:
    # Este código siempre se ejecuta si no
    # se ejecutó la sentencia break en el bloque for
```

Es decir, el código del bloque else se ejecutará siempre y cuando no se haya ejecutado la sentencia break dentro del bloque del `for`.

Ahora otro ejemplo de esto:

```
numeros = [1, 2, 4, 3, 5, 8, 6]

for n in numeros:
    if n == 3:
        break
else:
    print('No se encontró el número 3')
```

Como en el ejemplo anterior la secuencia números contiene al número `3`, la instrucción print nunca se ejecutará.

Conclusiones:

Con este capítulo pudimos aprender cómo funciona la estructura For en py, pues es bastante aplicable y operativa en diferentes problemas que tengas más adelante como profesionales.

Iterables y sus métodos:

Presentado por: Jose David Vazco Loaiza

Objetivo general: Profundizar en el concepto de iterable para py y conocer los distintos métodos que se pueden utilizar.

Resumen: En este capítulo vamos a analizar los diferentes métodos que existen en los objetos iterables tales como cadenas strings, listas, tuplas, diccionarios, etc… Así pues conoceremos su funcionalidad y aplicabilidad en diferentes áreas para resolver problemas de una forma más óptima.

Marco teórico:

Iterables y sus métodos.

Empezaremos por definir lo que es una lista. Las listas en Python son un tipo contenedor, compuesto, que se usan para almacenar conjuntos de elementos relacionados del mismo tipo o de tipos distintos.

Junto a las clases *tuple*, *range* e str, son uno de los tipos de secuencia en Python, con la particularidad de que son *mutables*. Esto último quiere decir que su contenido se puede modificar después de haber sido creado.

Ahora bien en el capítulo anterior vimos lo que es un iterable, retomando su concepto tenemos que, un iterable es un objeto que se puede iterar sobre él, es decir, que permite recorrer sus elementos uno a uno. Un iterador es un objeto que define un mecanismo para recorrer los elementos del iterable asociado.

Método de Strings:

- Método find(): Hay dos opciones para encontrar un subtring dentro de una string en Python, find() y rfind().

Cada uno retorna la posición en la que se encuentre la substring. La diferencia entre los dos, es que find() retorna la posición de la primera similitud de la substring y rfind() retorna la última posición de la similitud de la substring.

Nosotros podemos proveer algunos argumentos opcionales de inicio y fin para limitar la búsqueda de la substring, dentro de los limites del string.

Veamos un ejemplo:

```
>>> cadenaDeTexto = "Es peor cometer una injusticia que padecerla porque quien la comete se
>>> string.find('quien')
52
>>> string.rfind('quien')
94
```

- Método join(): El método str.join(iterable) es usado para unir todos los elementos de un iterable con un específico string str in Python. Si, el iterable no contiene ningún valor en los strings, Esto conduce a un TypeError exception.

veamos un ejemplo:

```
print(":".join(["freeCodeCamp", "es", "divertido"]))
```

- Método String replace: El mistr.replace(viejoString, nuevoString, numeroDeVeces) método es usado para reemplazar el substring: viejoString con el nuevoString que se reemplaza un numeroDeVeces . Éste método retorna una nueva copia del string reemplazado. El string original mistr no se ha modificado.

veamos un ejemplo:

```
string = "Esto es bonito. Esto es bueno."
newString = string.replace("es","FUE")
print(newString)
```

- Método String Strip: Hay 3 opciones para hacer eliminación de caracteres (stripping) de un string en Python: lstrip(), rstrip() y strip().

Cada uno retorna la posición del string con los caracteres removidos, al inicio y final o ambos. Si, no existen argumentos dados, por default para cortar o eliminar será un espacio en blanco.

Ahora veamos un ejemplo:

```
>>> string = '    Hola, Mundo!     '
>>> cortar_izquierda = string.lstrip()
>>> cortar_izquierda
'Hola, Mundo!     '
>>> cortar_derecha = string.rstrip()
>>> cortar_derecha
'    Hola, Mundo!'
>>> cortar_ambos_lados = string.strip()
>>> cortar_ambos_lados
'Hola, Mundo!'
```

- Método de string Split: Template: string.split(separador, maxdivision)

  separador: El delimitador del string. Tú divides el string basado en estos caracteres. Por ejemplo (" "), (":"), (";"), etc.

maxdivision: El número de veces a dividir el string basados en el separador. Si, no especificamos éste número o ponemos -1, el string tomará todas las occurrencias del separator.

Este método retorna una lista de substrings delimitados por el separator.

Ahora veamos un ejemplo:

```
string = "freeCodeCamp is fun."
print(string.split(" "))
```

```
['freeCodeCamp', 'is', 'fun.']
```

Nota: Si el separator no es especificado, Entonces el string se divide con el carácter por default el cual es " ". Tomará **todas** las ocurrencias del delimitador " ".

Métodos de listas:

- Metodo append(): Añade un ítem al final de la lista

Veamos un ejemplo:

```
lista = [1,2,3,4,5]
lista.append(6)
lista
```

```
[1, 2, 3, 4, 5, 6]
```

- Método clear: Vacía todos los ítems de una lista.

Veamos un ejemplo:

```
lista.clear()
lista
```

```
[]
```

- Método extend: Une una lista a otra.

Ejemplo:

```
l1 = [1,2,3]
l2 = [4,5,6]
l1.extend(l2)
l1
```

```
[1, 2, 3, 4, 5, 6]
```

- Método count(): Cuenta el número de veces que aparece un ítem.

Tenemos por ejemplo:

```
["Hola", "mundo", "mundo"].count("Hola")
```

```
1
```

- Método index: Devuelve el índice en el que aparece un ítem (error si no aparece)

Veamos un ejemplo:

```
["Hola", "mundo", "mundo"].index("mundo")
```

```
1
```

- Método insert(): Agrega un ítem a la lista en un índice específico. Como primera posición PY lo cuenta desde cero.

Ejemplo:

```
l = [1,2,3]
l.insert(0,0)
l
```

```
[0, 1, 2, 3]
```

- Método pop: Extrae un ítem de la lista y lo borra.

Ejemplo:

```
l = [10,20,30,40,50]
print(l.pop())
print(l)
```

```
50
[10, 20, 30, 40]
```

Métodos con tuplas:

- Método index: Index devuelve el número de índice del elemento que le pasemos por parámetro.

Ejemplo:

```
myTupla = (1,2,3)
myTupla.index(3)
```

- Método count: Para saber cuántas veces un elemento de una tupla se repite podemos utilizar el método count().

Ejemplo:

```
myTupla = (1,2,3,1,5,6,1)
myTupla.count(1)
# 3
```

Métodos de diccionarios:

- Método get(): Busca un elemento a partir de su clave y si no lo encuentra devuelve un valor por defecto.

veamos un ejemplo:

```
colores = { "amarillo":"yellow", "azul":"blue", "verde":"green" }
```

```
colores.get('negro','no se encuentra')
```

```
'no se encuentra'
```

- Método keys(): Genera una lista en clave de los registros del diccionario.

Ahora veamos un ejemplo.

```
colores.keys()
```

```
dict_keys(['amarillo', 'azul', 'verde'])
```

- Método values(): Genera una lista en valor de los registros del diccionario.

Ahora un ejemplo:

```
colores.values()
```

```
dict_values(['yellow', 'blue', 'green'])
```

- Método items(): Genera una lista en clave-valor de los registros del diccionario

Un ejemplo:

```
colores.items()
```

```
dict_items([('amarillo', 'yellow'), ('azul', 'blue'), ('verde', 'green')])
```

```
for clave, valor in colores.items():
    print(clave, valor)
```

```
amarillo yellow
azul blue
verde green
```

- Método pop(): Extrae un registro de un diccionario a partir de su clave y lo borra, acepta valor por defecto

Un ejemplo:

```
colores.pop("amarillo", "no se ha encontrado")
```

```
'yellow'
```

```
colores.pop("negro","no se ha encontrado")
```

```
'no se ha encontrado'
```

Conclusiones:

En este capítulo pudimos profundizar acerca de los conceptos de iterables y sus métodos y vimos la importancia que tiene este para la base de trabajo de python. Gracias a estas funciones y estos tipos de objetos que pueden almacenar datos es posible darle una aplicabilidad muy enorme a python, ahora bien también es importante recalcar que conocer el entorno python y su lenguaje de programación fue todo clave para poder adentrarnos esta vez un poco más en el mundo de matlab, para este caso aprendiendo de fondo las funciones que se pueden aplicar en cada objeto que se puede recorrer en python.