PROGRAMACIÓN PYTHON

UNIDAD 2 - Lazos e iteraciones.

Ejecución condicional: if, elif y else.

Como en cualquier lenguaje de programación, las estructuras de control en Python permiten cambiar el flujo de ejecución de los programas.

La palabra reservada *if* permite direccionar el flujo de ejecución del programa dependiendo de que alguna condición puede ser verdadera o falsa.

La sintaxis es la siguiente:

if *condición*:

> *acción1*
>
> *accion2*
>
> ...
>
> *accionM*

Como ejemplo práctico, hay que realizar un programa que resuelva ecuaciones de primer grado.

Probando collab

```
a = float(input("Valor de a:")) #lo que se leea de la consola se convertirá a float
b = float(input("Valor de b:")) #lo que se leea de la consola se convertirá a float

x = -b/a

print("Solución: ", x)
```

```
Valor de a:0
Valor de b:3
---------------------------------------------------------------------------
-
ZeroDivisionError                         Traceback (most recent call
last)
<ipython-input-3-f54a67365264> in <module>
      2 b = float(input("Valor de b:")) #lo que se leea de la consola se
convertirá a float
      3
----> 4 x = -b/a
      5
      6 print("Solución: ", x)

ZeroDivisionError: float division by zero
```

Para corregirlo se añadirán condiciones if para evaluar las posibles condiciones en las que puede incurrir el código. Probarlo con valores positivos, negativos, a en cero y b en cero.

NOTA: Observe las tabulaciones que están dentro de las condiciones if.

```
a = float(input("Valor de a:"))
b = float(input("Valor de b:"))

if a != 0:
  x = -b / a
  print("Solución: ", x)

if a == 0:
  if b != 0:
    print("La ecuación no tiene solución")
  if b == 0:
    print("La ecuación tiene soluciones infinitas")
```

```
Valor de a:2
Valor de b:3
Solución:  -1.5
```

Si es necesario ejecutar acciones distintas cuando la evaluación de la condición sea verdadera o falsa. Se puede utilizar la palabra reservada *else*.

```
a = float(input("Valor de a:"))
b = float(input("Valor de b:"))

if a != 0:
  x = -b / a
  print("Solución: ", x)

else:
  if b != 0:
    print("La ecuación no tiene solución")
  else:
    print("La ecuación tiene soluciones infinitas")
```

```
Valor de a:3
Valor de b:6
Solución:  -2.0
```

También es posible crear estructuras condicionales múltiples con la palabra reservada *elif*.

```
a = 0
b = 0
```

```
a = float(input("Valor de a:"))
b = float(input("Valor de b:"))

if a < b:
  print(a, " es menor que ", b)
elif a > b:
  print(a, " es mayor que ", b)
else:
  print(a," es igual que ", b)
```

```
Valor de a:4
Valor de b:6
4.0  es menor que  6.0
```

Operador ternario.

En Python existen también el operador ternario, se trata de una clúsula *if - else* que se define dentro de una sola línea, incluso puede ser utilizada dentro de un print(). La estructura es la siguiente:

[código si se cumple] if [condición] else [código si no se cumple]

```
x = 8
print("Es 5" if x == 5 else "No es 5")
```

También se puede utilizar para asignar valores a variables:

```
a = 10
b = 5
c = a / b if b != 0 else -1
print(c)
```

```
2.0
```

Iteraciones: while, for.

Ciclo for.

Este ciclo se caracteriza por tener un número de iteraciones definido de antemano. No tiene un candición de paro , más bien contiene un *iterable* que define las veces que se ejecutará el código. El siguiente ejemplo representa un ciclo representa un contador:

```
import time

for cont in range(6, 0, -1): #dentro de la función range: el primer número es el valor inicial de la vaiable, Segundo parámetro ha
  print("Valor de contador: " + str(cont)) #str convierte el valor numerico a string
  time.sleep(1)


print("Fin de ciclo...")
time.sleep(3)
```

```
    Valor de contador: 6
    Valor de contador: 5
    Valor de contador: 4
    Valor de contador: 3
    Valor de contador: 2
    Valor de contador: 1
    Fin de ciclo...
```

Iterables e iteradores

Los **iterables** son aquellos objetos que como su nombre indica pueden ser iterados, lo que dicho de otra forma es, que puedan ser indexados. Si piensas en una `list`, podemos indexarlo con `lista[1]` por ejemplo, por lo que sería un iterable. Algunos ejemplos son: listas, tuplas, cadenas o diccionarios.

Los **iteradores** son objetos que hacen referencia a un elemento, y que tienen un método `next` que permite hacer referencia al siguiente.

El ciclo for es capaz de recorrer los objetos iterables mediante la siguiente sintaxis:

```
 for <variable> in <iterable>:
     <Código>
```

Para saber si un objeto es iterable, se puede usar la función `isinstance()`:

```
#from para traer paquetes externos de programación
from collections.abc import  Iterable

lista = [1, 2, 3, 4]
cadena = "Python"
numero = 10
conjunto = {"platano", "manzana", "pera"}

print(isinstance(lista, Iterable)) #Si es iterable
print(isinstance(cadena, Iterable)) #Si es iterable
print(isinstance(numero, Iterable)) #No es iterable
print(isinstance(conjunto, Iterable)) #Si es iterable
```

```
    True
    True
    False
    True
```

La función `iter()` puede ser llamada desde un objeto que sea iterable y retorna un iterador, que es una variable que hace referencia al objeto iterable original y permite acceder a sus elementos con `next()`. Cuando coimienza, el iterador apunta afuera de la lista, y no hace referencia al primer elemento hasta que se llema a `next()` por primera vez.

```
it = iter(lista)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
#Mi modificación
print("Mostrará lista usando for")
```

```
it=iter(lista)
for it in range(1,5,1):
  print(it)
print("Mostrará cadena usando for")
for i in cadena:
  print(i)
```

```
1
2
3
4
1
2
3
4
P
y
t
h
o
n
```

Haz doble clic (o ingresa) para editar

Existen iteradores para diferentes clases:

`str_iterator` para cadenas.

`list_iterator` para listas.

`tuple_iterator` para tuplas.

`set_iterator` para sets.

`dict_keyiterator` para diccionarios.

El ciclo for, como ya se mencionó anteriormente, permite recorrer objetos iterables, ejemplos:

```
it = iter(lista)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
#Mi modificación
print("Mostrará lista usando for")
it=iter(lista)
for it in range(1,5,1):
  print(it)
print("Mostrará cadena usando for")
for i in cadena:
  print(i)
```

```
1
2
3
4
1
2
3
4
P
y
t
h
o
n
```

```
it = iter(lista)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
#Mi modificación
```

```
print("Mostrará lista usando for")
it=iter(lista)
for it in range(1,5,1):
  print(it)
print("Mostrará cadena usando for")
for i in cadena:
  print(i)
```

```
1
2
3
4
1
2
3
4
P
y
t
h
o
n
```

La función `range()` genera una secuencia de números que van desde 0 por defecto hasta el número que se pasa como parámetro menos 1. En realidad, se pueden pasar hasta tres parámetros separados por coma, donde el primer es el inicio de la secuencia, el segundo el final y el tercero el salto que se desea entre números. Por defecto se empieza en 0 y el salto es de 1:

```
range(inicio, fin, salto)
```

```
for i in range(5, 20, 3):
  print(i)
```

```
5
8
11
14
17
```

```
for i in range(5, 20, -1):
  print(i)
```

Edgar De La Rosa
10:03 Hoy
También es posible en forma descendente

Edgar De La Rosa
10:06 Hoy
Anidando ciclos

Los ciclos `for` también pueden anidarse, como en cualquier lenguaje de programación:

```
lista = [[56, 34, 1],
         [12, 4, 5],
         [9, 4, 2]] #esta lista simula un arreglo o matriz de 3 x 3
print(lista) #imprime cada lista
#IMprimire cada lista
for i in lista:
  print(i)
#For anidado
for i in lista:
  for j in i:
    print(j)

#print(i[0])
```

```
[[56, 34, 1], [12, 4, 5], [9, 4, 2]]
[56, 34, 1]
[12, 4, 5]
[9, 4, 2]
56
34
1
12
4
5
9
4
2
```

Ciclo while

EL ciclo while repetirá las sentencias dentro de él, mientras se cumpla la condición especificada.

```
#Hacer un contador con while
x = 0
while x < 6:
  print("Valor de contador: ", x)
  x += 1
  time.sleep(1)

print("Ciclo while terminado...")
time.sleep(3)
```

```
    Valor de contador:  0
    Valor de contador:  1
    Valor de contador:  2
    Valor de contador:  3
    Valor de contador:  4
    Valor de contador:  5
    Ciclo while terminado...
```

Edgar De La Rosa
10:17 Hoy

; se usa para indicar que en la misma linea o renglón: continua el código

También es posible escribir ciclos while en una sóla línea.

```
x = 6

while x > 0: print("Valor de contador: ", x); x -= 1

print("Fin de ciclo while")
```

```
    Valor de contador:  6
    Valor de contador:  5
    Valor de contador:  4
    Valor de contador:  3
    Valor de contador:  2
    Valor de contador:  1
    Fin de ciclo while
```

También permite recorrer algún elemento iterable mientras existan elementos en él, por ejemplo, una lista y en cada iteración se va eliminando un dato:

```
x = ["Uno", "Dos", "Tres"]
while x:
  x.pop(0) #obtener el elemento 0 de la lista
  print(x)
```

```
    ['Dos', 'Tres']
    ['Tres']
    []
```

Edgar De La Rosa
10:25 Hoy

while puede ser acompañado de else: permite validar si un ciclo se ejecutó correctamente

En Python, es posible mezclar la palabra reservada `else`, con el `while`. Servirá para ejecutar código una vez que el ciclo allá terminado normalmente:

```
x = 0
while x < 6:
  print("Valor de contador: ", x)
  x += 1
else:
  print("Fin de ciclo") #validar si el ciclo se ejecuto correctamente
```

```
    Valor de contador:  0
    Valor de contador:  1
    Valor de contador:  2
    Valor de contador:  3
    Valor de contador:  4
    Valor de contador:  5
    Fin de ciclo
```

Como el ciclo `while` ejecuta las sentencias en su interior cuando la condición es verdadera, se deberá tener cuidado de no generar ciclos infinitos, a menos que sea requerido por el programa a realizar. El

siguiente ejemplo es válido, pero ocasionará que el programa quedé "colgado" infinitamente:

```
while True:
  print("hola")
```

Modificación de ciclos.

`Break` permite detener la ejecución del ciclo. Una vez que el interprete encuentre la palabra, el ciclo terminará, sin embargo se puede considerar que es un final forzado.

Se puede utilizar tanto con ciclos `for` y `while`. Comúnmente se usa para establecer una condición de parada del ciclo y así evitar consumir ciclos de reloj de la computadora:

```
for letra in cadena:
  if letra == "h": #en python no existe los chars. Si uso comillas: siguen siendo un char
    print("Se encontró la h")
    break
  print(letra)
```

```
P
y
t
Se encontró la h
```

```
x = 5
while True:
  x -= 1
  print(x)

  if x == -5: #condicion de paro en caso de alguna anormalidad
    break
```

```
4
3
2
1
0
-1
-2
-3
-4
-5
```

El siguiente ejemplo muestra como un ciclo infinito se puede detener con una condición y un sentencia `break`:

```
for i in range(0,4):
  for j in range(0,4):
    break
  print(i, j)
```

```
0 0
1 0
2 0
3 0
```

Si `break` se encuentra dentro de ciclos anidados, únicamente detendŕa el ciclo en el que se encuentra:

La otra sentencia que permite modificar los ciclos en Python es la palabra reservada `continue`. Ésta permite que dentro de un ciclo se "salte" las líneas posteriores a donde se encuentra, haciendo que el ciclo continue su ejecución en la siguiente iteración:

```
#la palabra reservada continue: salta una iteración
for letra in cadena:
```

```
for letra in cadena:
  if letra == "y":
    continue
  print(letra)
print("\n")
for letra in cadena:
  if letra == "y" or letra == "t": #para que no considere dos caracteres
    continue
  print(letra)
print("\n")

for letra in cadena:
  if letra in "yt":               #para que no considere dos caracteres
    continue
  print(letra)
```

```
P
t
h
o
n


P
h
o
n


P
h
o
n
```

Como se puede observar, si bien se brinco el `print()`, únicamente no lo ejecuta en esa iteración, siendo que en las demás se ejecuta normalmente.

Un ejemplo con `while`:

COMPARTE TUS REFLEXIONES DE LOS TEMAS VISTOS ANTERIORMENTE.
