# Python - Clase 2

| | |
|---|---|
| Created | @August 26, 2022 5:00 PM |
| Lesson Date | @September 1, 2022 |
| Status | Complete |
| Type | Daily Lesson |

# Estructuras de control

## Condicional en Python

De no ser por las **estructuras de control**, el código en cualquier lenguaje de programación sería ejecutado secuencialmente hasta terminar. Un código, no deja de ser un conjunto de instrucciones que son ejecutadas unas tras otra. Gracias a las estructuras de control, podemos **cambiar el flujo de ejecución de un programa**, haciendo que ciertos bloques de código se ejecuten si y solo si se dan unas condiciones particulares.

### Uso del if

Un ejemplo sería si tenemos dos valores `a` y `b` que queremos dividir. Antes de entrar en el bloque de código que divide `a/b`, sería importante verificar que `b` es distinto de cero, ya que la división por cero no está definida. Es aquí donde entran los condicionales `if`.

```
a = 4
b = 2
if b != 0:
    print(a/b)
```

En este ejemplo podemos ver como se puede usar un `if` en Python. Con el operador `!=` se comprueba que el número `b` sea distinto de cero, y si lo es, se ejecuta el código que está identado. Por lo tanto un `if` tiene dos partes:

- La **condición** que se tiene que cumplir para que el bloque de código se ejecute, en nuestro caso `b!=0`.
- El **bloque de código** que se ejecutará si se cumple la condición anterior.

Es muy importante tener en cuenta que la sentencia `if` debe ir terminada por `:` y el bloque de código a ejecutar debe estar identado. Si usas algún editor de código, seguramente la identación se producirá automáticamente al presionar enter. Nótese que el bloque de código puede también contener más de una línea, es decir puede contener más de una instrucción.

```
if b != 0:
    c = a/b
    d = c + 1
    print(d)
```

Todo lo que vaya después del `if` y esté identado, será parte del bloque de código que se ejecutará si la condición se cumple. Por lo tanto el segundo `print()` “Fuera if” será ejecutado siempre, ya que está fuera del bloque `if`.

```
if b != 0:
    c = a/b
    print("Dentro if")
print("Fuera if")
```

Existen otros operadores que se verán en otros capítulos, como el de comparar si un número es mayor que otro. Su uso es igual que el anterior.

```
if b > 0:
    print(a/b)
```

Se puede también combinar varias condiciones entre el `if` y los `:`. Por ejemplo, se puede requerir que un número sea mayor que 5 y además menor que 15. Tenemos en realidad tres operadores usados conjuntamente, que serán evaluados por separado hasta devolver el resultado final, que será `True` si la condición se cumple o `False` de lo contrario.

```
a = 10
if a > 5 and a < 15:
    print("Mayor que 5 y menos que 15")
```

Es muy importante tener en cuenta que a diferencia de en otros lenguajes, en Python no puede haber un bloque `if` vacío. El siguiente código daría un `SyntaxError`.

```
if a > 5:
```

Por lo tanto si tenemos un `if` sin contenido, tal vez porque sea una tarea pendiente que estamos dejando para implementar en un futuro, es necesario hacer uso de `pass` para evitar el error. Realmente `pass` no hace nada, simplemente es para tener contento al interprete de código.

```
if a > 5:
    pass
```

## Uso de else y elif

Es posible que no solo queramos hacer algo si una determinada condición se cumple, sino que además queramos hacer algo de lo contrario. Es aquí donde entra la cláusula `else`. La parte del `if` se comporta de la manera que ya hemos explicado, con la diferencia que si esa condición no se cumple, se ejecutará el código presente dentro del `else`. Nótese que ambos bloque de código son excluyentes, se entra o en uno o en otro, pero nunca se ejecutarán los dos.

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

Hasta ahora hemos visto como ejecutar un bloque de código si se cumple una instrucción, u otro si no se cumple, pero no es suficiente. En muchos casos, podemos tener varias condiciones diferentes y para cada una queremos un código distinto. Es aquí donde entra en juego el `elif`.

```
x = 5
if x == 5:
    print("Es 5")
elif x == 6:
    print("Es 6")
elif x == 7:
    print("Es 7")
```

Con la cláusula `elif` podemos ejecutar tantos bloques de código distintos como queramos según la condición. Traducido al lenguaje natural, sería algo así como decir: si es igual a 5 haz esto, si es igual a 6 haz lo otro, si es igual a 7 haz lo otro.

Se puede usar también de manera conjunta todo, el `if` con el `elif` y un `else` al final. Es muy importante notar que `if` y `else` solamente puede haber uno, mientras que `elif` puede haber varios.

```
x = 5
if x == 5:
    print("Es 5")
elif x == 6:
    print("Es 6")
elif x == 7:
    print("Es 7")
else:
    print("Es otro")
```

Si vienes de otros lenguajes de programación, sabrás que el `switch` es una forma alternativa de `elif`, sin embargo en Python esta cláusula no existe.

## Ejemplos if

```
# Verifica si un número es par o impar
x = 6
if not x%2:
    print("Es par")
else:
    print("Es impar")
```

```
# Decrementa x en 1 unidad si es mayor que cero
x = 5
```

```
x-=1 if x>0 else x
print(x)
```

# Bucles

## Bucle for

A continuación explicaremos el bucle `for` y sus particularidades en Python, que comparado con otros lenguajes de comparación, tiene ciertas diferencias.

El for es un tipo de bucle, parecido al while pero con ciertas diferencias. La principal es que el número de iteraciones de un for **esta definido** de antemano, mientras que en un while no. La diferencia principal con respecto al while es en la condición. Mientras que en el while la condición era evaluada en cada iteración para decidir si volver a ejecutar o no el código, en el `for` no existe tal condición, sino un `iterable` que define las veces que se ejecutará el código. En el siguiente ejemplo vemos un bucle `for` que se ejecuta 5 veces, y donde la `i` incrementa su valor “automáticamente” en 1 en cada iteración.

```
for i in range(0, 5):
    print(i)

# Salida:
# 0
# 1
# 2
# 3
# 4
```

```
for i in "Python":
    print(i)

# Salida:
# P
# y
# t
# h
# o
# n
```

## For anidados

Es posible **anidar** los `for`, es decir, **meter uno dentro de otro**. Esto puede ser muy útil si queremos iterar algún objeto que en cada elemento, tiene a su vez otra clase iterable. Podemos tener por ejemplo, una lista de listas, una especie de matriz.

```
lista = [[56, 34, 1],
         [12, 4, 5],
         [9, 4, 3]]
```

Si iteramos usando sólo un `for`, estaremos realmente accediendo a la segunda lista, pero no a los elementos individuales.

```
for i in lista:
    print(i)
#[56, 34, 1]
#[12, 4, 5]
#[9, 4, 3]
```

Si queremos acceder a cada elemento individualmente, podemos anidar dos `for`. Uno de ellos se encargará de iterar las columnas y el otro las filas.

```
for i in lista:
    for j in i:
        print(j)
# Salida: 56,34,1,12,4,5,9,4,3
```

## Ejemplos for

Iterando cadena al revés. Haciendo uso de `[::-1]` se puede iterar la lista desde el último al primer elemento.

```
texto = "Python"
for i in texto[::-1]:
    print(i) #n,o,h,t,y,P
```

Itera la cadena saltándose elementos. Con `[::2]` vamos tomando un elemento si y otro no.

```
texto = "Python"
for i in texto[::2]:
    print(i) #P,t,o
```

```
print(sum(i for i in range(10)))

# Salida: 45
```

## Uso del range

Uno de las iteraciones mas comunes que se realizan, es la de iterar un número entre por ejemplo 0 y `n`. Si ya programas, estoy seguro de que estas cansado de escribir esto, aunque sea en otro lenguaje. Pongamos que queremos iterar una variable `i` de 0 a 5. Haciendo uso de lo que hemos visto anteriormente, podríamos hacer lo siguiente.

```
for i in (0, 1, 2, 3, 4, 5):
    print(i) #0, 1, 2, 3, 4, 5
```

Se trata de una solución que cumple con nuestro requisito. El contenido después del `in` se trata de una clase que como ya hemos visto antes, es iterable, y es de hecho una tupla. Sin embargo, hay otras formas de hacer esto en Python, haciendo uso del `range()`.

```
for i in range(6):
    print(i) #0, 1, 2, 3, 4, 5
```

El `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)`

Por lo tanto, si llamamos a `range()` con `(5,20,2)`, se generarán números de 5 a 20 de dos en dos. Un truco es que el `range()` se puede convertir en `list`.

`print(list(range(5, 20, 2)))`

Y mezclándolo con el `for`, podemos hacer lo siguiente.

```
for i in range(5, 20, 2):
    print(i) #5,7,9,11,13,15,17,19
```

Se pueden generar también secuencias inversas, empezando por un número mayor y terminando en uno menor, pero para ello el salto deberá ser negativo.

```
for i in range (5, 0, -1):
    print(i) #5,4,3,2,1
```

## While

El uso del `while` nos permite **ejecutar una sección de código repetidas veces**, de ahí su nombre. El código se ejecutará **mientras** una condición determinada se cumpla. Cuando se deje de cumplir, se saldrá del bucle y se continuará la ejecución normal. Llamaremos **iteración** a una ejecución completa del bloque de código.

Cabe destacar que existe dos tipos de bucles, los que tienen un número de iteraciones **no definidas**, y los que tienen un número de iteraciones **definidas**. El `while` estaría dentro del primer tipo. Mas adelante veremos los `for`, que se engloban en el segundo.

```
x = 5
while x > 0:
    x -=1
    print(x)

# Salida: 4,3,2,1,0
```

En el ejemplo anterior tenemos un caso sencillo de `while`. Tenemos una condición `x>0` y un bloque de código a ejecutar mientras dure esa condición `x-=1` y `print(x)`. Por lo tanto mientras que `x` sea mayor que 0, se ejecutará el código. Una vez se llega al final, se vuelve a empezar y si la condición se cumple, se ejecuta

otra vez. En este caso se entra al bloque de código 5 veces, hasta que en la sexta, `x` vale cero y por lo tanto la condición ya no se cumple. Por lo tanto el `while` tiene dos partes:

- La **condición** que se tiene que cumplir para que se ejecute el código.
- El **bloque de código** que se ejecutará mientras la condición se cumpla.

Ten cuidado ya que un mal uso del `while` puede dar lugar a bucles infinitos y problemas. Cierto es que en algún caso tal vez nos interese tener un bucle infinito, pero salvo que estemos seguros de lo que estamos haciendo, hay que tener cuidado. Imaginemos que tenemos un bucle cuya condición siempre se cumple. Por ejemplo, si ponemos `True` en la condición del `while`, siempre que se evalúe esa expresión, el resultado será `True` y se ejecutará el bloque de código. Una vez llegado al final del bloque, se volverá a evaluar la condición, se cumplirá, y vuelta a empezar. No te recomiendo que ejecutes el siguiente código, pero puedes intentarlo.

```
# No ejecutes esto, en serio
while True:
    print("Bucle infinito")
```

También podemos usar otro tipo de operación dentro del `while`, como la que se muestra a continuación. En este caso tenemos una lista que mientras no este vacía, vamos eliminando su primer elemento.

```
x = ["Uno", "Dos", "Tres"]
while x:
    x.pop(0)
    print(x)
#['Dos', 'Tres']
#['Tres']
#[]
```

## Else y while

Algo no muy corriente en otros lenguajes de programación pero si en Python, es el uso de la cláusula `else` al final del `while`. Podemos ver el ejemplo anterior mezclado con el `else`. La sección de código que se encuentra dentro del `else`, se ejecutará cuando

el bucle termine, pero solo si lo hace “por razones naturales”. Es decir, si el bucle termina porque la condición se deja de cumplir, y no porque se ha hecho uso del `break`.

```
x = 5
while x > 0:
    x -=1
    print(x) #4,3,2,1,0
else:
    print("El bucle ha finalizado")
```

Podemos ver como si el bucle termina por el `break`, el `print()` no se ejecutará. Por lo tanto, se podría decir que si no hay realmente ninguna sentencia `break` dentro del bucle, tal vez no tenga mucho sentido el uso del `else`, ya que un bloque de código fuera del bucle cumplirá con la misma funcionalidad.

```
x = 5
while True:
    x -= 1
    print(x) #4, 3, 2, 1, 0
    if x == 0:
        break
else:
    # El print no se ejecuta
    print("Fin del bucle")
```

## Bucles anidados

Ya hemos visto que los bucles `while` tienen una condición a evaluar y un bloque de código a ejecutar. Hemos visto ejemplos donde el bloque de código son operaciones sencillas como la resta `-`, pero podemos complicar un poco mas las cosas y meter otro bucle `while` dentro del primero. Es algo que resulta especialmente útil si por ejemplo queremos generar permutaciones de números, es decir, si queremos generar todas las combinaciones posibles. Imaginemos que queremos generar todas las combinaciones de de dos números hasta 2. Es decir, 0-0, 0-1, 0-2,… hasta 2-2.

```
# Permutación a generar
i = 0
j = 0
while i < 3:
    while j < 3:
        print(i,j)
```

```
        j += 1
    i += 1
    j = 0
```

Vamos a analizara el ejemplo paso por paso. El primer bucle genera números del 0 al 2, lo que corresponde a la variable `i`. Por otro lado el segundo bucle genera también número del 0 al 2, almacenados en la variable `j`. Al tener un bucle dentro de otro, lo que pasa es que por cada `i` se generan 3 `j`. Muy importante no olvidar que al finalizar el bucle de la `j`, debemos resetear `j=0` para que en la siguiente iteración la condición de entrada se cumpla.Podemos complicar las cosas aún más y tener tres bucles anidados, generando combinaciones de 3 elementos con número 0, 1, 2. En este caso tendremos desde 0,0,0 hasta 2,2,2.

## Ejemplos while

Árbol de navidad en Python. Imprime un árbol de navidad formado con `*` haciendo uso del `while` y de la multiplicación de un entero por una cadena, cuyo resultado en Python es replicar la cadena.

```
z = 7
x = 1
while z > 0:
    print(' ' * z + '*' * x + ' ' * z)
    x+=2
    z-=1
#       *
#      ***
#     *****
#    *******
#   *********
# ***********
```

Aunque esta no sea tal vez la mejor forma de iterar una cadena es un buen ejemplo para el uso del `while` e introducir el indexado de listas con `[]`, que veremos en otros capítulos.

```
text = "Python"
i = 0
while i < len(text):
    print(text[:i + 1])
    i += 1
# P
```

```
# Py
# Pyt
# Pyth
# Pytho
# Python
```

Sucesión de **Fibonacci** en Python. En matemáticas, la sucesión de *fibonacci* es una sucesión infinita de números naturales, donde el siguiente es calculado sumando los dos anteriores.

```
a, b = 0, 1
while b < 25:
    print(b)
    a, b = b, a + b
#1, 1, 2, 3, 5, 8, 13, 21
```

# Sentencia break Python

La sentencia `break` nos permite alterar el comportamiento de los bucles while y for. Concretamente, permite terminar con la ejecución del bucle.

Esto significa que una vez se encuentra la palabra `break`, el bucle se habrá terminado.

## Break con bucles for

Veamos como podemos usar el `break` con bucles for. El `range(5)` generaría 5 iteraciones, donde la `i` valdría de 0 a 4. Sin embargo, en la primera iteración, terminamos el bucle prematuramente.

El `break` hace que nada más empezar el bucle, se rompa y se salga sin haber hecho nada.

```
for i in range(5):
    print(i)
    break
    # No llega

# Salida: 0
```

Un ejemplo un poco más útil, sería el de buscar una letra en una palabra. Se itera toda la palabra y en el momento en el que se encuentra la letra que buscábamos, se rompe

el bucle y se sale.

Esto es algo muy útil porque si ya encontramos lo que estábamos buscando, no tendría mucho sentido seguir iterando la lista, ya que desperdiciaríamos recursos.

```
cadena = 'Python'
for letra in cadena:
    if letra == 'h':
        print("Se encontró la h")
        break
    print(letra)

# Salida:
# P
# y
# t
# Se encontró la h
```

## Break con bucles while

El `break` también nos permite alterar el comportamiento del while. Veamos un ejemplo.

La condición `while True` haría que la sección de código se ejecutara indefinidamente, pero al hacer uso del `break`, el bucle se romperá cuando `x` valga cero.

```
x = 5
while True:
    x -= 1
    print(x)
    if x == 0:
        break
    print("Fin del bucle")

#4, 3, 2, 1, 0
```

Por norma general, y salvo casos muy concretos, si ves un `while True`, es probable que haya un `break` dentro del bucle.

## Break y bucles anidados

Como hemos dicho, el uso de `break` rompe el bucle, pero sólo aquel en el que está dentro.

Es decir, si tenemos dos bucles anidados, el `break` romperá el bucle anidado, pero no el exterior.

```
for i in range(0, 4):
    for j in range(0, 4):
        break
        #Nunca se realiza más de una iteración
    # El break no afecta a este for
    print(i, j)

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

# Ejercicios

1. Realizar un menú de un cajero automático, donde el usuario pueda escoger entre alguna de las siguientes opciones:
   - Deposito.
   - Extracción.
   - Transferencia.
   - Salir.

   En todos los casos se le pedirá al usuario ingresar un monto de dinero y el programa deberá mostrar en todo momento la sección del menú en la que se encuentre, pudiendo retornar al menú principal siempre que se quiera y solo se detendrá la ejecución cuando se ingrese la opción de “salir” en el menú principal.

2. Idear un programa que solicite al usuario dos números enteros y el programa deberá retornar aquellos números pares que se encuentren dentro del rango formado entre los números ingresados.