Introducción a la programación y al análisis de datos con Python

Programación básica

Índice

Esquema

Ideas clave

3.1. Introducción y objetivos

Este tema introducirá nuevos conceptos que nos permitirán añadir complejidad a nuestros programas. En primer lugar, enumeraremos las estructuras de datos que nos proporciona Python para almacenar objetos más complejos. A continuación, explicaremos las sentencias condicionales que nos permitirán ejecutar diferentes bloques de código dependiendo de situaciones o resultados. Por último, explicaremos las formas que tenemos de ejecutar bloques de código de manera repetida usando los bucles y, además, veremos los iteradores que nos permiten recorrer secuencias de elementos.

Al finalizar este tema, habrás alcanzado los siguientes objetivos:

- Conocer las diferentes estructuras de datos incluidas en Python.
- Aprender las funciones más importantes para cada una de las estructuras de datos.
- Comprender cómo hacer bifurcaciones en la ejecución de los programas con las sentencias condicionales.
- Conocer las sentencias de los bucles que nos permiten ejecutar bloques de código repetidamente.
- Saber utilizar los iteradores para recorrer secuencias de valores.

3.2. Estructuras de datos

En el tema anterior vimos los tipos de datos básicos que podemos utilizar en Python para almacenar valores. Sin embargo, este tipo de datos no son suficientes para almacenar valores complejos como, por ejemplo, una lista de usuarios de una página web. En este tema, veremos las estructuras de datos que nos proporciona Python para almacenar valores más complejos.

Listas

Las listas son las estructuras de datos más utilizadas en Python. Una lista es una estructura que nos permite tener un conjunto de objetos separados por comas. Esta estructura de datos es **mutable,** es decir, podemos cambiar el valor de una lista que hemos creado como, por ejemplo, cambiar el orden de los elementos, eliminar elementos, etc.

En una lista pueden existir elementos de diferentes tipos, es decir, elementos con diferentes tipos de datos o, incluso, diferentes estructuras de datos. Sin embargo, lo normal es que todos los elementos de una lista sean del mismo tipo. Para declarar una lista, escribimos entre corchetes ([]) un conjunto de elementos separados por comas:

```
lista = [3, 'Hola', True]
```

También podemos crear una lista vacía usando únicamente los corchetes sin ningún valor dentro o utilizando la función `list()`:

```
lista_vacia = []
```

A partir de una lista que hayamos creado, podemos acceder a los distintos elementos de dicha lista. Para ello, solo tenemos que indicar la posición que ocupa el elemento

dentro de la lista, es decir, indicar el *índice del elemento*. Debemos tener en cuenta que el primer elemento de la lista ocupa la posición 0.

```
lista = [3, 'Hola', True]
lista[2] # Devolverá el valor True
```

También podemos acceder a un conjunto de elementos de una lista usando el símbolo de dos puntos (:) dentro de los corchetes. A esto se le llama un rango de índices. Por ejemplo, para acceder a las 2 primeras posiciones de una lista podríamos hacer lo siguiente:

```
lista = [3, 'Hola', True]
lista[:2] # Devolverá [3, 'Hola']
```

Cuando usamos los rangos de índices, como el visto anteriormente, existen unos valores por defecto cuando no indicamos su valor. El valor por defecto del primer índice es 0 y el valor por defecto del último índice es la longitud de la lista.

```
lista = [3, 'Hola', True]
lista[:2] # Devolverá [3, 'Hola']
lista[1:] # Devolverá ['Hola', True]
```

Además, dentro de los índices de las listas, ya sea un solo índice o un rango, podemos usar valores negativos. En esos casos Python devuelve los elementos contando su posición desde la derecha.

```
lista = [3, 'Hola', True]
lista[-1] # Devolverá True
lista[-2:] # Devolverá ['Hola', True]
```

Funciones aplicables a listas

Cuando usamos listas podemos usar un conjunto de funciones que nos permiten obtener propiedades de las listas o modificarlas. A continuación, vamos a enumerar algunas de las funciones más utilizadas en listas:

- **`len`**: devuelve la longitud de una lista, es decir, el número de elementos incluidos en una lista.

```
lista = [3, 'Hola', True]
len(lista) # Devolverá 3
```

- **`index`**: devuelve la posición que ocupa un elemento dentro de una lista.

```
lista = [3, 'Hola', True]
lista.index('Hola') # Devolverá 1
```

- **`insert`**: inserta un elemento dentro de una lista en la posición que le indicamos.

```
lista = [3, 'Hola', True]
lista.insert(1, 'Adiós')
lista # Nos mostrará [3, 'Adiós', 'Hola', True]
```

- **`append`**: inserta un elemento al final de la lista. En el caso de que pongamos una lista de elementos, la función lo insertará como un elemento único.

```
lista = [3, 'Hola', True]
lista.append([3, 4])
lista # Nos mostrará [3, 'Adiós', 'Hola', True, [3, 4]]
```

- **`extend`**: permite agregar un conjunto de elementos en una lista. A diferencia del método anterior, si incluimos una lista de elementos, se agregarán cada uno de los elementos a la lista.

```
lista = [3, 'Hola', True]
lista.extend([3, 4])
lista # Nos mostrará [3, 'Adiós', 'Hola', True, 3, 4]
```

- **remove**: elimina el elemento que pasamos por parámetro de la lista. En caso de que este elemento estuviese repetido, solo se eliminará la primera copia.

```
lista = [3, 'Hola', True, 'Hola']
lista.remove('Hola')
lista # Nos mostrará [3, True, 'Hola']
```

- **count**: devuelve el número de veces que se encuentra un elemento en una lista.

```
lista = [3, 'Hola', True, 'Hola']
lista.count('Hola')  # Nos devolverá 2
```

- **reverse**: este método nos permite invertir la posición de todos los elementos de la lista.

```
lista = [3, 'Hola', True]
lista.reverse()
lista # Nos mostrará [True, 'Hola', 3]
```

- **sort**: ordena los elementos de una lista. Por defecto, este método los ordena en orden creciente. Para ordenarlo de forma decreciente, hay que incluir el parámetro (`reverse=True`). ¡Ojo! La lista debe contener elementos del mismo tipo.

```
lista = [6, 4, 1, 9, 7, 0, 5]
lista.sort()
lista # Nos mostrará [0, 1, 4, 5, 6, 7, 9]
lista.sort(reverse=True)
lista # Nos mostrará [9, 7, 6, 5, 4, 1, 0]
```

- **pop**: elimina y devuelve el elemento que se encuentra en la posición que se pasa por parámetro. En caso de que no se pase ningún valor por parámetro, eliminará y devolverá el último elemento de la lista.

```
lista = [6, 4, 1, 9, 7, 0, 5]
lista.pop(1) # Devolverá 4
lista # Nos mostrará [6, 1, 9, 7, 0, 5]
```

Tuplas

Al igual que las listas, las tuplas son conjuntos de elementos separados por comas. Sin embargo, a diferencia de las listas, las tuplas son **inmutables,** es decir, no se pueden modificar una vez creadas. Un ejemplo de tupla sería el siguiente:

```
tupla = 'Hola', 3.4, True, 'Hola'
tupla
```

En el ejemplo anterior, hemos creado una tupla a partir de una secuencia de valores separados por comas. El resultado es una tupla que contiene todos estos elementos. A esta operación se le denomina **empaquetado de tuplas**. También disponemos del método inverso. Si a una tupla de longitud n le asignamos n variables, cada una de las variables tendrá uno de los componentes de la tupla. A esta operación se le llama **desempaquetado de tuplas**.

```
w, x, y, z = tupla # w = 'Hola', x = 3.4, y = True, z = 'Hola'
```

Para acceder a los elementos de una tupla, lo haremos de la misma manera que con las listas:

```
tupla[1] # Nos mostrará 3.4
tupla[1:] # Nos mostrará (3.4, True, 'Hola')
```

Funciones aplicables a tuplas

Al igual que las listas, tenemos varios métodos que podemos usar en las tuplas. Los métodos más utilizados son los siguientes:

- **`len`**: método que devuelve la longitud de la tupla.

```
tupla = 'Hola', 3.4, True, 'Hola'
len(tupla) # Devolverá 4
```

- **count**: número de veces que se encuentra un elemento en una tupla.

```
tupla.count('Hola') # Devolverá 2
```

- **index**: devuelve la posición que ocupa un elemento dentro de una tupla. En caso de que el elemento este repetido, devolverá la primera posición donde aparece el objeto.

```
tupla.index('Hola') # Devolverá 0
```

Diccionarios

La última estructura de datos que veremos en Python son los diccionarios. Los diccionarios conforman una estructura que enlaza los elementos almacenados con claves *(keys)* en lugar de índices, como las estructuras anteriores. Es decir, para acceder a un objeto es necesario hacerlo a través de su clave.

La mejor manera de comprender un diccionario es verlo como un conjunto de pares (clave, valor), donde las claves son únicas, es decir, no están repetidas, y nos permiten acceder al objeto almacenado. Para crear un diccionario definiremos un conjunto de elementos clave valor delimitados por llaves ({}):

```
diccionario = {
      'clave1': 'Mi primer valor',
      'clave3': 'Y, como no, mi tercer valor',
      'clave2': 'Este es mi segundo valor'
}
```

Si queremos acceder a uno de sus valores, necesitaremos conocer su clave y lo pondremos entre corchetes ([]):

```
diccionario['clave1'] # Devolverá 'Mi primer valor'
```

Para crear nuevos elementos en los diccionarios usamos la misma forma de acceso a un elemento, pero asignando un nuevo valor:

```
diccionario['clave_nueva'] = ‘nuevo valor’
```

Podemos crear diccionarios vacíos usando las llaves, pero sin insertar ningún elemento, o usando la función `dict()`:

```
diccionario = dict()
```

Podemos eliminar un elemento del diccionario usando la instrucción `del` e indicando qué elemento del diccionario queremos eliminar.

```
del diccionario['clave1'] # Eliminará el elemento con clave 'clave1'
```

Funciones aplicables a diccionarios

A continuación, enumeraremos dos de las funciones más utilizadas en los diccionarios.

- **`list`**: devuelve una lista de todas las claves incluidas en un diccionario. Si queremos la lista ordenada, usaremos la función **`sorted`** en lugar de **`list`**.

  ```
  list(diccionario) # Devolverá ['clave1', 'clave3', 'clave2']
  sorted(diccionario) # Devolverá ['clave1', 'clave2', 'clave3']
  ```

- **`in`**: comprueba si una clave se encuentra en el diccionario.

  ```
  'clave3' in diccionario # Devolverá True
  ```

Conjuntos

Los conjuntos son colecciones no ordenadas de elementos. Además, los conjuntos no contienen repetición de elementos, es decir, se eliminan todos los elementos duplicados. Para crear un conjunto podemos indicar un conjunto de objetos separados por comas y delimitados por llaves {}.

```
conjunto = {'audi', 'mercedes', 'seat', 'ferrari', 'ferrari', 'renault'}
```

Para crear un conjunto vacío podemos crearlo usando las llaves sin insertar ningún valor o la función set().

```
conjunto = set()
conjunto # Nos mostrará {}
```

Al no ser una lista ordenada, no podemos acceder a sus elementos a través de su índice; esto nos devolverá un error de tipo.

Funciones aplicables a conjuntos

A continuación, veremos algunas de las operaciones soportadas por los conjuntos de Python.

- **union**: operación matemática para obtener la unión de dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
conjunto1.union(conjunto2) # Devolverá {1, 2, 3, 4, 5}
```

- **intersection**: operación matemática para obtener la intersección de dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
conjunto1.intersection(conjunto2) # Devolverá {3}
```

- **difference**: operación matemática para obtener la diferencia del conjunto original con respecto al conjunto que se pasa por parámetro, es decir, los elementos del primer conjunto que no están en el segundo.

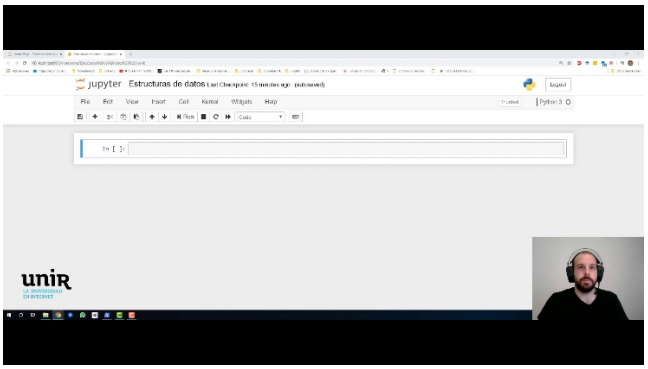
```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
conjunto1.difference(conjunto2) # Devolverá {1, 2}
```

- **in**: comprueba si un elemento se encuentra dentro de un conjunto.

```
1 in conjunto1 # Devolverá True
```

- **len**: devuelve la longitud del conjunto.

```
len(conjunto1) # Devolverá 3
```



Vídeo 1. Estructuras de datos.

Accede al vídeo a través del aula virtual

3.3. Ejecuciones condicionales

Hasta este momento durante el curso hemos visto algunas sentencias que nos permiten hacer pequeños programas en Python. Sin embargo, en la mayoría de las ocasiones los programas que haremos no tendrán una ejecución lineal, sino que habrá bifurcaciones dependiendo del resultado que obtengamos de algunas evaluaciones. Para poder hacer estas bifurcaciones en la ejecución de nuestro programa es necesario utilizar las ejecuciones condicionales.

Expresión `if`

Las expresiones `if` nos permiten ejecutar un bloque de instrucciones únicamente si la expresión lógica que hemos puesto devuelve `True`. Para escribir una sentencia `if` se sigue la siguiente estructura:

```
if (EXPRESION_LOGICA):
    sentencia_1
    sentencia_2
    ...
```

Aquí tenemos un ejemplo de cómo podríamos usar una sentencia `if`. En este ejemplo pedimos al usuario que escriba un valor entre 1 y 10. A continuación, comprobamos si el número es mayor que 5 y, si es así, imprimimos un mensaje:

```
print("Escribe un número de 1 a 10")

number = int(input())
if (number > 5):
    print("¡Soy mayor que 5!")
```

Expresión `else`

Por otro lado, si queremos ejecutar otro bloque de instrucciones cuando no se cumple la condición del `if`, usaremos la expresión `else`. Esta expresión debe ir siempre después del bloque de instrucciones del `if`:

```
if (EXPRESION_LOGICA):
    sentencia_1
    sentencia_2
    ...
else:
    sentencia_1
    sentencia_2
    ...
```

Podemos ampliar el ejemplo anterior para que, en el caso de que el número no sea mayor que 5, mostremos otro mensaje al usuario. Para ello, incluiremos un bloque `else` después del bloque `if`:

```
print("Escribe un número de 1 a 10")

number = int(input())
if (number > 5):
    print("¡Soy mayor que 5!")
else:
    print("Soy menor o igual que 5")
```

Expresión `elif`

La expresión `elif` nos permite evaluar más condiciones dentro de un `if` para ejecutar otros bloques de instrucciones. Si incluimos un bloque `else`, las instrucciones de este

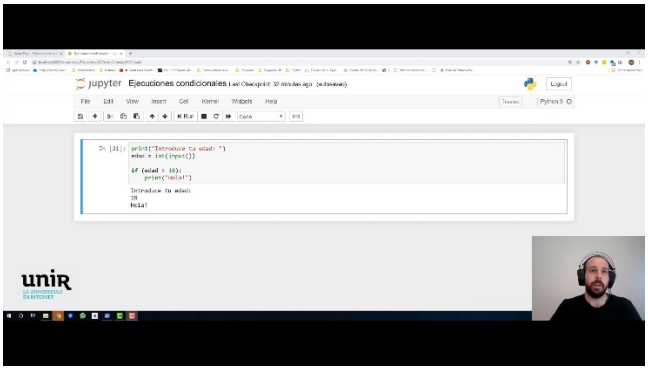
bloque solo si todas las expresiones lógicas de los bloques `if` y `elif` han devuelto `False`. Se puede añadir más de un bloque `elif` y estos bloques deben ir después del bloque `if` y antes del bloque `else`:

```
if (EXPRESION_LOGICA):
    sentencia_1
    ...
elif (EXPRESION_LOGICA_2):
    sentencia_1
    ...
elif (EXPRESION_LOGICA_3):
    sentencia_1
    ...
...
else:
    sentencia_1
    sentencia_2
    ...
```

Vamos a aplicar esta expresión en nuestro ejemplo anterior. Para ello, comprobaremos también si el valor introducido por el usuario es 5 y, si es así, mostraremos otro mensaje:

```
print("Escribe un número de 1 a 10")

number = int(input())
if (number > 5):
    print("¡Soy mayor que 5!")
elif (number == 5):
    print("Soy el número 5")
else:
    print("Soy menor o igual que 5")
```



Vídeo 2. Ejecuciones condicionales.

Accede al vídeo a través del aula virtual

3.4. Ejecuciones iterativas

En esta sección vamos a explicar las dos expresiones que existen en Python para ejecutar varias veces un conjunto de instrucciones. Esas expresiones son la sentencia `while` y la sentencia `for`. Además, veremos otras sentencias que se pueden utilizar en los bucles para modificar el flujo de ejecución y, por último, veremos el uso de iteradores que nos permiten recorrer todos los elementos de objetos como las listas.

Bucle `while`

La primera instrucción que vamos a explicar en las iteraciones es `while`. Esta instrucción repite un bloque de código mientras se cumpla una condición definida por nosotros. Esa condición, al igual que pasaba con `if`, viene dada en forma de expresión lógica. El bloque de instrucciones de la instrucción `while` dejará de ejecutarse cuando esa expresión lógica devuelva un `False`. El formato de escritura de `while` es el siguiente:

```
while (EXPRESION_LOGICA):
    sentencia_1
    sentencia_2
    ...
```

Por ejemplo, imaginemos que queremos imprimir una secuencia de números desde el número 5 hasta el número 0. Esto lo podríamos hacer con un bloque `while`, al cual, mientras el número que estamos imprimiendo sea mayor que 0, le restaremos una unidad y lo imprimiremos:

```
numero = 5
fin = 0

while(numero > fin):
    numero -= 1
    print(numero)
```

A la instrucción `while` se la puede incluir la sentencia `else`. A diferencia de las ejecuciones condicionales, el bloque de instrucciones de la sentencia `else` se

ejecutará siempre cuando acabe de ejecutarse las iteraciones en `while`. En este caso, escribiríamos la sentencia `else` a continuación de las instrucciones del bloque `while`.

```
while (EXPRESION_LOGICA):
    sentencia_1
    sentencia_2
    ...
else:
    sentencia_1
    sentencia_2
    ...
```

En el ejemplo anterior, podemos imprimir un mensaje al usuario para indicarle que hemos acabado de imprimir números. Para ello, incluimos un bloque `else` después del bloque `while` con la instrucción que imprimirá el mensaje:

```
numero = 5
fin = 0

while(numero > fin):
    numero -= 1
    print(numero)
else:
    print("Ya he acabado!")
```

Esta es la forma más sencilla de crear sentencias iterativas. Normalmente, la instrucción `while` se utiliza para hacer búsquedas de elementos o ejecutar un conjunto de acciones hasta que ocurre un evento. En ambos casos, no se sabe exactamente cuántas ejecuciones tenemos que hacer y depende de la evaluación de una expresión lógica.

Bucle `for`

La segunda forma de crear secuencias iterativas es con la instrucción `for`. Esta permite recorrer un conjunto de elementos (por ejemplo, listas) en cualquier sentido y con cualquier paso. La forma general de crear una sentencia `for` es la siguiente:

```
for VARIABLE in OBJETO:
    sentencia_1
    sentencia_2
    ...
```

```
else:
    sentencia_1
    sentencia_2
    ...
```

En la instrucción `for` declaramos una variable a la que, en cada iteración del bucle, se le asignará el valor de uno de los elementos contenidos en `OBJETO`. Como pasaba en la sentencia `while`, podemos incluir una sentencia `else` que ejecutará sus instrucciones cuando hayan finalizado todas las iteraciones del bucle `for`.

Podemos hacer el mismo ejemplo que en la sentencia `while`. Para ello creamos una lista con los números que queremos imprimir. Con la instrucción `for` recorremos cada uno de esos números y los imprimimos. Al final, mostraremos un mensaje indicando que hemos acabado:

```
numeros = [5, 4, 3, 2, 1, 0]

for numero in numeros:
    print(numero)
else:
    print("Ya he acabado!")
```

Esta forma de asignación también se puede hacer con cadenas de texto. En este caso, a la variable se le irá asignando cada uno de los caracteres de la cadena en cada paso del bucle:

```
texto = 'Hola mundo'

for caracter in texto:
    print(caracter)
```

Además, como es lógico, la variable puede tener un tipo de dato diferente en cada vuelta del bucle.

```
lista = ['texto', 5, (23, 56)]

for elemento in lista:
    print(elemento)
```

En las sentencias `for`, una forma de recorrer un objeto, como una lista, es a través de sus índices. El objetivo es que la variable tenga como valor en cada paso la posición de la lista que estamos visitando. Para poder hacer esto, en Python se utiliza la instrucción `range`. Esta instrucción nos devuelve un rango de números desde un número inicial hasta uno final, y con la separación entre números que hayamos seleccionado. Su estructura es una de las siguientes:

```
# Secuencia de números de 0 a NUMERO_FINAL con paso 1
range(NUMERO_FINAL)

# Secuencia de números de NUMERO_INICIAL a NUMERO_FINAL con paso 1
range(NUMERO_INICIAL, NUMERO_FINAL)

# Secuencia de números de NUMERO_INICIAL a NUMERO_FINAL con paso PASO
range(NUMERO_INICIAL, NUMERO_FINAL, PASO)
```

Los rangos se pueden utilizar directamente como objeto en el bucle `for`. Sin embargo, para poder imprimir todos los índices que nos ha generado un rango, es necesario encapsular el rango en una lista.

A continuación, vemos un ejemplo del uso de rangos. En este ejemplo solo queremos mostrar los caracteres que ocupan una posición par en la cadena «Hola mundo», podríamos hacerlo así:

```
cadena = 'Hola mundo'

# Comenzamos en 0, hasta la longitud de la cadena y con paso = 2
for i in range(0, len(cadena), 2):
    print(cadena[i])
```

Sentencias extras

A las estructuras de iteración que hemos visto antes podemos incluir otras sentencias que permiten modificar la ejecución de los bucles. Estas sentencias se pueden utilizar tanto en los bucles `while` como en los bucles `for`.

- **`break`**: esta sentencia rompe la ejecución del bucle en el momento en que se ejecute.

```
numeros = list(range(10))

for n in numeros:
    if (n == 5):
        print('Rompemos el bucle!')
        break

    print(n)

# La secuencia de ejecución sería: 0, 1, 2, 3, 4, Rompemos el bucle!
```

- **`Continue`**: esta instrucción permite saltarnos una iteración del bucle sin que se rompa la ejecución final.

```
numeros = list(range(10))

for n in numeros:
    if (n == 5):
        print('Me salto una vuelta')
        continue

    print(n)

# La secuencia de ejecución sería: 0, 1, 2, 3, 4, Me salto una vuelta,
6, 7, 8, 9
```

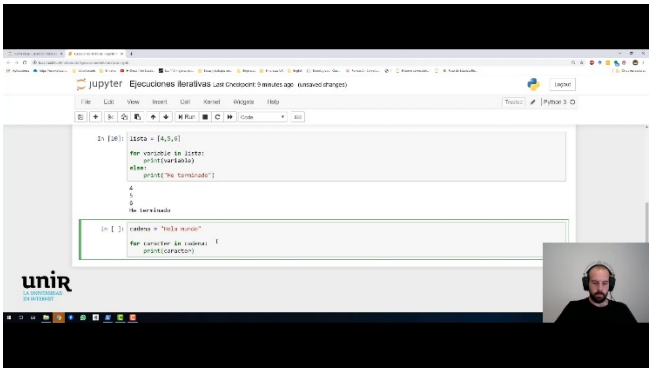
Iteradores

Otra forma de recorrer los elementos de un objeto en Python es utilizando iteradores. Un iterador es un objeto que, al aplicarlo sobre otro objeto iterable, como son las cadenas de texto o los conjuntos, nos permite obtener el siguiente elemento por visitar.

Para crear un iterador sobre un objeto usamos la instrucción **`iter`**`(OBJETO)` y se lo asignamos a una variable. Una vez creado, podemos visitar los elementos del `OBJETO` con la función **`next`**`()` y pasando por parámetro el identificador del iterador. Veremos esto en un ejemplo.

```
cadena = 'Hola'
iterador = iter(cadena)

next(iterador) # Devolverá 'H'
next(iterador) # Devolverá 'o'
next(iterador) # Devovlerá 'l'
next(iterador) # Devolverá 'a'
```



Vídeo 3. Ejecuciones iterativas.

Accede al vídeo a través del aula virtual