Introducción a la programación y al análisis de datos con Python

Aspectos avanzados

Índice

Esquema

Ideas clave

6.1. Introducción y objetivos

Este es el último tema que trata la introducción a la programación en Python. En este tema veremos algunas sentencias más avanzadas para aplicar a nuestros programas. En primer lugar, describiremos el uso de las expresiones regulares para buscar patrones dentro de textos. A continuación, veremos cómo se generan los errores en Python y qué forma tenemos de gestionarlos para que no se termine la ejecución. Por último, explicaremos el uso de compresión de listas, una forma sencilla de crear listas cuyos elementos se obtienen al aplicar una operación a los elementos de otra secuencia.

Al finalizar este tema habremos conseguido los siguientes objetivos:

- Conocer las funciones para buscar y modificar elementos en textos.
- Aprender la sintaxis de las expresiones regulares para definir patrones de texto.
- Aplicar estructuras que nos permitan controlar los errores que podemos tener en nuestros programas.
- Crear diferentes estructuras de datos usando la técnica de compresión.

6.2. Expresiones regulares

En un tema anterior vimos algunas funciones dentro de las cadenas de caracteres que nos permitían obtener los caracteres que ocupan posiciones concretas o conocer la posición de subcadenas que ya conocemos. Sin embargo, en muchas ocasiones necesitamos buscar otras subcadenas que no tienen el mismo contenido siempre,

pero que siguen un patrón, como un número de teléfono. Para poder buscar este tipo de subcadenas necesitamos utilizar expresiones regulares.

Las expresiones regulares son una secuencia de caracteres con otros caracteres especiales que nos permiten definir patrones de texto. Utilizando estas expresiones regulares, podemos buscar estos patrones en cadenas de texto.

Para aplicar las expresiones regulares en cadenas de texto, usaremos el módulo `re` de Python. Para ello, importaremos este módulo la primera vez que lo utilicemos:

```
import re
```

A continuación, vamos a explicar algunos de los métodos que tenemos disponibles en `re`:

- **`search()`:** esta función busca la primera aparición del patrón en la cadena de texto. Como resultado nos devuelve un objeto de tipo `match` en el que podemos obtener las posiciones donde se encuentra el patrón dentro de la cadena de texto. En caso de que no exista el patrón dentro de la cadena, la función devuelve un objeto `None`.

```
mensaje = "Esto es un mensaje de prueba para el curso de Python."
match = re.search('curso', mensaje)
print("Comienzo:", match.start(), "Final:", match.end()) # Devolverá
Comienzo: 37 Final: 42
```

- **`match()`:** busca un patrón al principio de la cadena. En caso de que no exista el patrón o no se encuentre al principio de la cadena, devolverá un objeto de tipo `None`.

```
mensaje = "Esto es un mensaje de prueba para el curso de Python."
match = re.match('Esto', mensaje)
print("Comienzo:", match.start(), "Final:", match.end()) # Devolverá
Comienzo: 0 Final:4
```

- **`split()`:** este método nos permite dividir una cadena de caracteres siguiendo un patrón. Como resultado nos devuelve una lista con cada una de las divisiones.

```
mensaje = "Esto es un mensaje de prueba para el curso de Python."
re.split(' ', mensaje) # Devolverá una lista donde cada elemento es una palabra del mensaje.
```

- **`sub()`:** permite sustituir los patrones encontrados por otra subcadena que pasamos por parámetro.

```
mensaje = "Esto es un mensaje de prueba para el curso de Python."
re.sub('Python', 'Java', mensaje) # Devolverá 'Esto es un mensaje de prueba para el curso de Java.'
```

- **`findall()`:** busca todas las apariciones de un patrón dentro de una cadena de caracteres. Como resultado nos devolverá una lista con todas las subcadenas que cumplen con el patrón.

```
mensaje = "Esto es un mensaje de prueba para el curso de Python."
re.findall('de', mensaje) # Devolverá ['de', 'de']
```

Estas son las funciones más importantes. Sin embargo, ahora es necesario conocer cómo construir patrones más complejos que nos permitan buscar subcadenas con diferente contenido, pero con quien tengan un patrón en común, por ejemplo, fechas de nacimiento, apellidos, localidades, etc. Para los siguientes ejemplos usaremos un texto más amplio extraído de Wikipedia:

```
texto = "Un texto es una composición de signos codificados en un sistema de escritura que forma una unidad de sentido.

También es una composición de caracteres imprimibles (con grafema) generados por un algoritmo de cifrado que, aunque no tiene sentido para cualquier persona, sí puede ser descifrado por su destinatario original. En otras palabras, un texto es un entramado de signos con una intención comunicativa que adquiere sentido en determinado contexto."
```

Para comprender las expresiones regulares, es necesario conocer los distintos componentes que tenemos dentro de una de ellas:

- **Literales:** estos son los elementos que contienen únicamente caracteres básicos. Son los que hemos visto en los ejemplos anteriores para explicar las funciones del módulo `re`.

- **Caracteres de escape:** se utilizan para definir caracteres especiales dentro de una cadena de texto como, por ejemplo, son los saltos de línea. También es necesario utilizarlos para buscar caracteres que dentro de una expresión regular tienen un significado propio, por ejemplo, el asterisco (*). Los caracteres de escape comienzan con una barra invertida (\). Algunos de los caracteres de escape más importantes son los siguientes:

  - **`\n`:** salto de línea.
  - **`\t`:** tabulador.
  - **`\\`:** barra diagonal inversa.
  - **`\d`:** un dígito.
  - **`\w`:** un carácter alfanumérico.
  - **`\s`:** un espacio en blanco.
  - **`\D`:** carácter que no sea un dígito.
  - **`\W`:** carácter que no sea alfanumérico.

  Para usarlos, solo debemos incluirlos dentro de una cadena de texto que usaremos en la función correspondiente del módulo `re`. Por ejemplo, si queremos sustituir todos los saltos de línea por el texto "`**AQUÍ**`" en el texto, aplicaremos el carácter de escape correspondiente para usarlo en la función `sub`:

```
re.sub('\n', '**AQUI**', texto)
```

El resultado será el siguiente texto:

```
'Un texto es una composición de signos codificados en un sistema de escritura que forma una unidad de sentido.**AQUI****AQUI**También es una composición de caracteres imprimibles (con grafema) generados por un algoritmo de cifrado que, aunque no tiene sentido para cualquier persona, sí puede ser descifrado por su destinatario original. En otras palabras, un texto es un entramado de signos con una intención comunicativa que adquiere sentido en determinado contexto.'
```

- **Grupos de caracteres:** El uso de grupos de caracteres nos permiten, no solo encontrar un patrón en el texto, sino también capturar el resultado del patrón para procesarlo más adelante.

Vamos a ver esto a través de un ejemplo. Imaginemos que queremos capturar las palabras que empiecen por *se-* y que tengan otra palabra a continuación. Para ello podemos usar el siguiente patrón:

```
pattern = 'se\w+\s\w+'
re.findall(pattern, texto) # Devolverá ['sentido para', 'ser descifrado', 'sentido en']
```

Sin embargo, nosotros solo queremos conocer cuál es la palabra que empieza por *se-* y no el resto de los elementos, es decir, la palabra que viene a continuación. Para ello, agrupamos el patrón que define las palabras que empiezan por *se-* usando los paréntesis:

```
pattern = '(se\w+)\s\w+'
re.findall(pattern, texto) # Devolverá ['sentido', 'ser', 'sentido']
```

Como se puede apreciar, esto nos permite capturar únicamente la parte que nos interesa del patrón, es decir, las palabras que empiezan por *se-* pero que, además, tienen otra palabra a continuación.

- **Metacaracteres:** los metacaracteres son caracteres que tienen un significado especial dentro de las expresiones regulares. Estos metacaracteres permiten buscar repeticiones de patrones, tipos de caracteres, etc. A continuación, describiremos algunos de los metacaracteres más comunes:

  - **|:** nos permite separar distintas alternativas que estamos buscando dentro de un texto. Por ejemplo, si queremos sustituir cualquier aparición de las palabras *es* o *ser* por `**AQUÍ**`, usaremos el siguiente patrón:

    ```
    pattern = 'es|ser'
    re.sub(pattern, '**AQUI**', texto)
    ```

  - **?:** el elemento que le precede aparece una vez o ninguna. Imaginemos, en el siguiente ejemplo, que buscamos cualquier número que tenga un dígito o ninguno seguido de un espacio:

    ```
    ejemplo = '1 22 333 4444'
    pattern = '\d?\s'
    re.findall(pattern, ejemplo) # Devolverá ['1 ', '2 ', '3 ']
    ```

  - **+:** el elemento que le precede aparece una o más veces. Por ejemplo, en el siguiente ejemplo, vamos a buscar el número donde aparezca el dígito 3 una o más veces:

    ```
    ejemplo = '1 22 333 4444'
    pattern = '3+'
    re.findall(pattern, ejemplo) # Devolverá ['333']
    ```

  - ***:** el elemento que le precede aparece ninguna o más veces. Por ejemplo, en el siguiente ejemplo buscaremos qué parte contiene el dígito 1 ninguna o más veces seguido del número 2:

    ```
    ejemplo = '122333114444'
    pattern = '(1*2)'
    re.findall(pattern, ejemplo) # Devolverá ['12', '2']
    ```

- **{n}:** el elemento que le precede aparece n veces. En el siguiente ejemplo buscaremos qué parte de la cadena de caracteres tiene exactamente 3 dígitos:

```
ejemplo = '1 22 333 4444'
pattern = '\d{3}'
re.findall(pattern, ejemplo) # Devolverá ['333', '444']
```

- **{n,m}:** el elemento anterior aparece entre n y m. Si n está vacío significará que el elemento aparece de 0 a m veces. Si, por el contrario, m es vacío significará que el elemento aparece n o más veces. A continuación, buscaremos qué parte de la cadena de texto tiene 2 o 3 dígitos seguidos:

```
ejemplo = '1 22 333 4444'
pattern = '\d{2,3}'
re.findall(pattern, ejemplo) # Devolverá ['22', '333', '444']
```

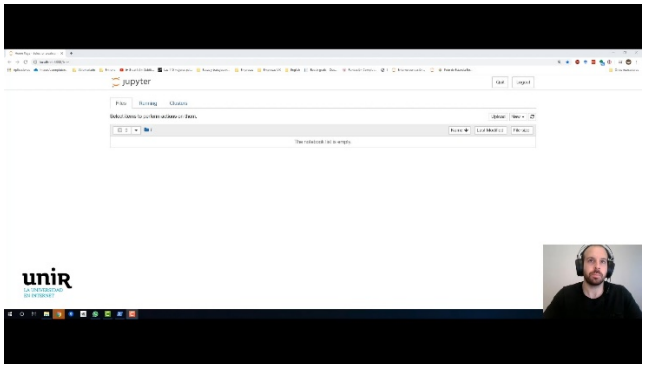
- **[]:** nos permite representar clases de caracteres, es decir, buscará cadenas que tengan algunos de los caracteres definidos dentro de los corchetes. Por ejemplo, a continuación, buscaremos qué parte de la cadena tiene los dígitos 2 o 3 repetidos entre 2 y 3 veces:

```
ejemplo = '1 22 333 4444'
pattern = '[2,3]{2,3}'
re.findall(pattern, ejemplo) # Devolverá ['22', '333']
```

- **-:** nos permite definir un rango de caracteres. En el ejemplo siguiente, buscaremos qué parte de la cadena tiene los valores entre 1 y 3 repetidos 2 o más veces.

```
ejemplo = '1 22 333 4444'
pattern = '[1-3]{2,}'
re.findall(pattern, ejemplo) # Devolverá ['22', '333']
```

Estos son los métodos y elementos más importantes en las expresiones regulares. Para profundizar mucho más puedes recurrir a la documentación que indicamos en la sección A fondo.



Vídeo 1. Expresiones regulares.

Accede al vídeo a través del aula virtual

6.3. Errores y excepciones

En este apartado vamos a explicar qué son los errores y las excepciones de un programa y qué herramientas nos proporciona Python para gestionarlos.

Errores

En ocasiones, cuando estamos ejecutando un programa, pueden ocurrir errores de ejecución. Normalmente, estos errores detienen la ejecución del programa y muestran un mensaje en la consola, donde se nos indica el tipo de error ocurrido y en qué parte del código. Este mensaje se muestra de la siguiente forma en Jupyter Notebook.

```
texto = 'Hola'
numero = 3
texto + numero
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-51f4cde6d8bf> in <module>
      1 texto = 'Hola'
      2 numero = 3
----> 3 texto + numero

TypeError: can only concatenate str (not "int") to str
```

Figura 1. Error de ejecución (tipo).

En este caso, Python nos devuelve un error de tipo `TypeError`, es decir, un error asociado a los tipos de datos. El error se ha producido exactamente en la tercera línea y, además, nos muestra un mensaje que explica por qué se ha producido ese error. En este caso, Python nos indica que solo se pueden concatenar tipos *string,* no enteros, que es lo que estamos intentando hacer en este caso.

Los errores más comunes que nos podemos encontrar en nuestros programas son los siguientes:

- **Errores de tipo:** son como los que acabamos de ver. Se producen cuando no podemos realizar una operación porque el tipo de datos de alguno de los elementos no es válido.

- **Errores de sintaxis:** estos errores se producen cuando una sentencia o instrucción no está bien escrita y Python nos indica que tenemos un error en la sintaxis de la sentencia. Por ejemplo, no cerrar un paréntesis:

```
print('Hola'
```

```
  File "<ipython-input-3-84df3280101f>", line 1
    print('Hola'
                ^
SyntaxError: unexpected EOF while parsing
```

Figura 2. Error de sintaxis.

- **Errores de nombre:** se producen cuando llamamos a un identificador que Python no sabe cuál es. Por ejemplo, si escribimos mal un identificador o introducimos una función que no hemos declarado anteriormente:

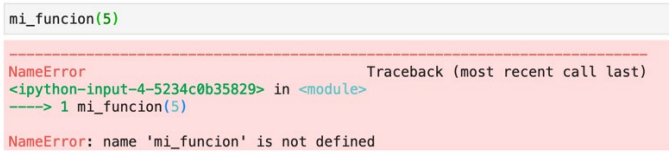


Figura 3. Error de nombre.

- **Errores de índice:** estos errores aparecen cuando trabajamos con secuencias, como listas, e intentamos acceder a una posición que no existe. Por ejemplo, si intentamos acceder a la primera posición de una lista vacía:

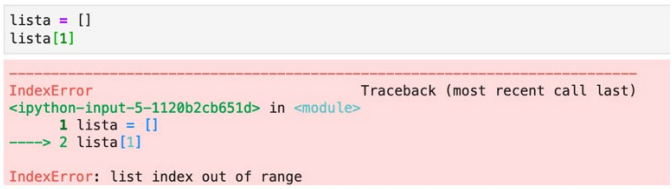


Figura 4. Error de índice.

Existen otros muchos errores que podemos tener al ejecutar nuestros programas. Sin embargo, estos son los que encontraremos con más frecuencia.

Excepciones

Para alguno de estos errores es interesante que termine la ejecución, ya que son errores críticos que imposibilitan que continuemos con nuestro programa. Pero, en otras ocasiones, podemos asumir que un error de cierto tipo puede ocurrir en algún momento y no queremos que se termine la ejecución del programa.

Para poder hacer esto utilizaremos las excepciones. Una excepción es un bloque de código que se ejecutará cuando se detecte un error de algún tipo y, después, podremos seguir ejecutando nuestro programa. Para poder hacer esto, es necesario

utilizar los bloques **try** junto con los bloques de código **except**. El esquema de nuestro código quedaría de la siguiente manera:

```
try:
    bloque_con_posibles_errores
except:
    bloque_de_excepcion
```

En primer lugar, usamos la sentencia `try` y, a continuación, escribimos el bloque de instrucciones en el que sabemos que puede ocurrir un error. A continuación, y con la misma sangría que la sentencia `try`, escribimos la sentencia `except` seguido del bloque de instrucciones que se ejecutará cuando ocurra un error en el bloque try. Vamos a ver su funcionamiento a través de un ejemplo:

```
lista = [2, 3, 4]

try:
    print(lista[5])
except:
    print("No existe esa posición")

print("Sigo con la ejecución")
```

En este código accederemos a la posición de una lista, pero queremos ejecutar una excepción si esa posición no existe. En esa excepción se nos mostrará un mensaje, pero luego seguirá con la ejecución del programa. El sistema nos mostrará los siguientes mensajes al ejecutarse:

```
No existe esa posición
Sigo con la ejecución
```

Vemos cómo se ha detectado un error, en este caso, un error de índice, y se ha ejecutado la instrucción que aparece en el bloque `except`. A continuación, ha seguido con la ejecución del programa.

Es posible crear más de un bloque `except` para que nos permitan ejecutar diferentes bloques de código para diferentes tipos de errores. Para ello incluiremos el tipo de

error. En el siguiente ejemplo, vamos a preparar diferentes bloques para diferentes tipos de errores:

```
lista = [2, 3, 4]

try:
    print(lista[5])
except IndexError:
    print("No existe esa posición")
except TypeError:
    print("El tipo de la operación es erróneo")
except:
    print("Ha ocurrido otro error")

print("Sigo con la ejecución")
```

Como se puede observar, hemos incluido una sentencia final, sin definir el tipo de error. Este bloque se ejecutará si ha ocurrido un error, pero no se ha ejecutado ninguno de los otros bloques `except`. A la estructura `try-except` se le pueden incluir más tipos de bloques, como veremos a continuación.

Bloque `else`

Igual que podemos ejecutar bloques cuando se ha detectado un error, también podemos ejecutar otro bloque de código cuando no ha habido ningún error. Para ello usamos el bloque **`else`:**

```
try:
    bloque_con_posibles_errores
except:
    bloque_de_excepcion
else:
    bloque_else
```

Vamos a ampliar el ejemplo anterior para que, en el caso de que no se haya detectado ningún error, se muestre otro mensaje. Para ello también cambiaremos la posición a la que accedemos.

```
lista = [2, 3, 4]

try:
    print(lista[2])
except:
    print("No existe esa posición")
else:
    print("Está perfecto")

print("Sigo con la ejecución")
```

En este ejemplo, la consola de Python nos devolverá el siguiente resultado:

```
4
Está perfecto
Sigo con la ejecución
```

En primer lugar, nos devuelve el valor que hay en la isla en la posición 2. A continuación, como no ha habido ningún error, ejecuta el bloque de la sentencia `else`, que en este caso es un mensaje diciendo que está perfecto. Por último, continua con la ejecución del sistema.

Bloque `finally`

Otro tipo de bloque que se puede incluir en la estructura `try-except` es el bloque `finally`. Este bloque se ejecuta siempre, haya habido o no un error en el bloque `try`. La estructura completa `try-except` con todos los bloques sería como se muestra a continuación:

```
try:
    bloque_con_posibles_errores
except:
    bloque_de_excepcion
else:
    bloque_else
finally:
    bloque_finally
```

Ampliaremos el ejemplo anterior añadiendo un bloque `finally` que imprima un mensaje donde se nos indique que hemos terminado de ejecutar el bloque `try-except`:

```
lista = [2, 3, 4]

try:
    print(lista[2])
except:
    print("No existe esa posición")
else:
    print("Está perfecto")
finally:
    print("Hemos terminado el bloque try-except")

print("Sigo con la ejecución")
```

El resultado de ejecutar este ejemplo será el siguiente:

```
4
Está perfecto
Hemos terminado el bloque try-except
Sigo con la ejecución
```

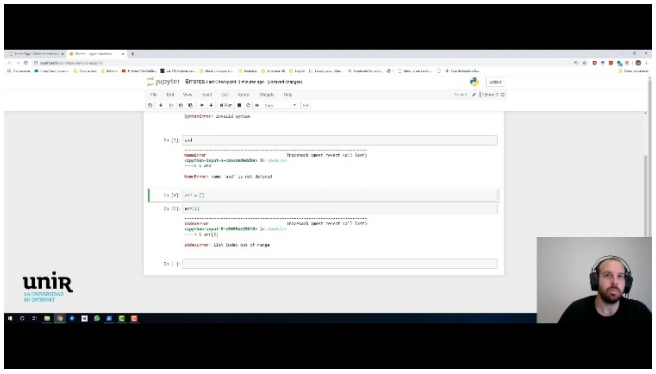
El bloque `finally` se utiliza mucho para cerrar ficheros o conexiones a bases de datos, se haya producido o no un error, para asegurarnos que cerramos dicha conexión.

Lanzar errores

En ocasiones, es útil para nosotros lanzar un error para que se gestione en otro lado de nuestro programa. Para ello, usaremos la sentencia `raise`, seguida del tipo de error y el mensaje que queremos mostrar. Vamos a ver el uso de esta sentencia en el siguiente ejemplo:

```
try:
    raise TypeError("Este es un ejemplo de error personalizado")
except:
    print("He detectado un error")
```

Esto es útil para las funciones que desarrollemos, ya que podemos lanzar errores si los argumentos que nos han introducido no son válidos o la ejecución de la función no se puede realizar por cualquier motivo.



Vídeo 2. Errores y excepciones.

Accede al vídeo a través del aula virtual

6.4. Compresión de listas

La compresión de listas es una forma que nos proporciona Python que nos permite crear listas donde cada uno de los miembros de la lista es el resultado de una operación. Por ejemplo, imaginemos que queremos crear una lista que almacene el resultado de aplicar el cuadrado a los números de 1 a 10. Usando un bucle `for`, lo haríamos de la siguiente manera:

```
result = []
for i in range(1, 11):
    result.append(i ** 2)
```

Sin embargo, esta operación se puede realizar de una forma más sencilla gracias a la compresión de listas. La estructura de una compresión de listas es como:

```
[operación for ítem in secuencia]
```

Como vemos en este esquema tenemos tres elementos. El primero de ellos es `operación`, que será una expresión que devuelve un resultado y cuyo resultado se almacenará en la lista resultante. A continuación, tenemos un ítem de la secuencia de elementos que se van a recorrer. Normalmente, los ítems son utilizados en las operaciones. Por último, tenemos una secuencia iterable que tendrá todos los valores de entrada que queremos recorrer para generar la lista.

En el ejemplo anterior, cada uno de los elementos serían: `i**2` (operación), `i` (ítem), `range(1,11)` (secuencia). Es decir, podemos aplicar compresión de listas en el ejemplo anterior de la siguiente manera:

```
result = [i**2 for i in range(1,11)]
```

Como se puede observar, hemos hecho la misma acción, pero en este caso lo hemos implementado en una única línea. El uso de compresión de listas hace que el código sea más compacto y, sobre todo, que la ejecución sea mucho más rápida.

A esta estructura se le pueden incluir expresiones condicionales que se deben cumplir para que un elemento pueda ser incluido en la lista resultante. El esquema de la compresión de listas con expresiones condicionales sería la siguiente:

```
[operación for item in secuencia if condicion]
```

Por ejemplo, podemos hacer que una lista almacene el resultado de aplicar el cuadrado únicamente a los números múltiplos de 3 que existan desde 1 a 10:

```
result = [i**2 for i in range(1,11) if i % 3 == 0]
```

Estas sentencias nos permiten crear nuevas listas de una forma muy sencilla. Sin embargo, la compresión puede ser aplicada a otros tipos de colecciones. A continuación, explicaremos cómo aplicar a otras colecciones compatibles.

Diccionarios

Para los diccionarios, es necesario que definamos la clave y el valor de cada elemento que insertemos en ellos. Por este motivo, la expresión de compresión para diccionarios debe incluir ambos conceptos, el resto de la expresión es igual que la anterior:

```
{clave: valor for item in secuencia if condicion}
```

Por ejemplo, vamos a crear un diccionario que almacena el resultado de aplicar el cuadrado a los números pares que hay de 1 a 10, pero donde la clave de dichos resultados sea el número con el que se calculó el cuadrado:

```
result = {i: i**2 for i in range(1,11) if i % 2 == 0}
```

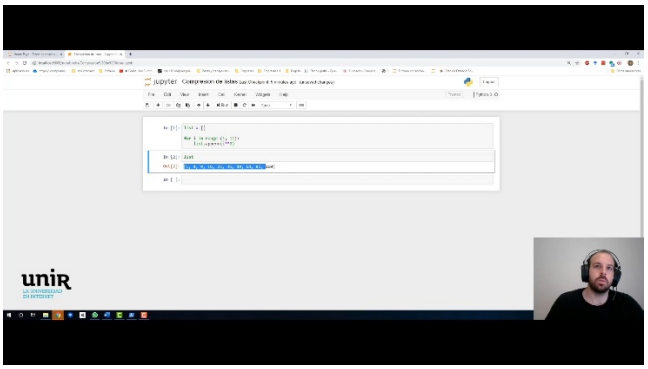
Conjuntos

Al igual que pasa con los diccionarios, tenemos que adaptar la expresión de compresión para que se creen conjuntos. En este caso, usaremos los símbolos de creación de conjuntos, seguidos de la misma estructura de compresión que usábamos con las listas:

```
{item for item in secuencia if condicion}
```

Por ejemplo, si queremos crear una secuencia que almacena el resultado de aplicar el cuadrado a los números pares que hay de 1 a 10, aplicaríamos la siguiente sentencia:

```
result = {i**2 for i in range (1,11) if i % 2 == 0}
```



Vídeo 3. Compresión de listas.

Accede al vídeo a través del aula virtual