# SESSION 2. ESTRUCTURAS DE DATOS Y FUNCIONES

Victor Miguel Terrón Macias

19/2/2021

## PREWORK. ESTRUCTURAS DE DATOS Y FUNCIONES

### LISTAS

Datos, datos, datos... Están en todas partes. Evidentemente, nosotros queremos aprender a analizarlos, así que entendemos su importancia. Cuando pensamos en datos solemos pensar en bases de datos, en archivos CSV, en hojas de Excel, y otros. Cada uno de estos tiene su propia estructura de datos. Las computadoras necesitan estas estructuras de datos para organizar los datos de manera eficiente. Sin estructuras de datos, tendríamos todos nuestros datos "regados por ahí". Encontrar lo que estamos buscando sería imposible. Y todos nuestros procesos se harían de la forma más ineficiente posible. El día de hoy conoceremos dos de las estructuras de datos más básicas e importantes que tiene Python: las listas y los diccionarios. Listas Las listas son colecciones ordenadas de elementos. ¿De qué elementos hablamos? Bueno, en el caso de Python, prácticamente de lo que sea. ¿Recuerdas los tipos de datos que aprendimos en la sesión anterior? Pues cualquiera de esos puede ser incluido dentro de una lista. Si tenemos una serie de elementos que queremos guardar de forma ordenada, podemos meterlos dentro de una lista. Para crear una lista, usamos corchetes ([]) y luego colocamos nuestros elementos dentro de los corchetes. Así:

```
In [1]: soy_una_lista_feliz = [3, 6, 1, 4, 8, 7, 2, 4, 3]
```

Figure 1: IMAGEN 1

Como puedes ver, el orden lo determino yo. Es decir, los números no están ordenados de menor a mayor. Yo decido en qué orden ponerlos y así se quedan. También podemos observar que es posible repetir elementos idénticos en diferentes lugares de la lista. En este caso hicimos una lista de ints, pero, como ya dijimos, podemos tener listas de prácticamente cualquier cosa:

```
In [1]: lista_de_floats = [1.4, 5.78, 2.34, 5.21, 8.9]

        lista_de_strings = ["Hola", "mundo", "me", "llamo", "Marco P."]

        lista_de_booleanos = [True, False, True, True, False, False]

        lista_de_listas_de_ints = [[1, 4, 3], [6, 8, 7], [2, 4, 3], [0, 9, 8]]
```

Figure 2: IMAGEN 2

Sí, ¡incluso podemos tener listas de listas! ¡O listas de listas de listas! ¡O listas de listas de listas de listas! ¡O...! Ok, ya. Como puedes observar, todas las listas que te acabo de mostrar tienen el mismo de dato. Es posible en Python hacer listas con diferentes tipos de datos. Se ven así:

Aunque en principio es posible, mezclar diferentes tipos de dato en una misma lista es una mala idea. Al hacerlo estamos haciendo más complicado procesar los datos que tenemos en esa lista. Es decir, va en contra

```
In [2]: lista_mixta = [3, True, 7.5, [3, 5, 4], "Hola"]
```

Figure 3: IMAGEN 3

de una de las razones principales que tenemos para usar estructuras de datos: la eficiencia. A veces es inevitable, pero debemos de evitarlo siempre que sea posible. Ok, ya tengo mi lista. ¿Ahora, cómo accedo a los elementos que tiene dentro? Toda lista tiene un índice ligado a cada elemento que la conforma. Como las listas son ordenadas, el índice empieza con 0 para el primer elemento, y luego va ascendiendo de uno en uno para los elementos subsecuentes. Podemos revisar la lista usando el operador de índice: [] . Lo usamos pasándole el índice del elemento al que quiero acceder:

```
In [3]: lista_1 = [4, 8, 3, 6, 2]

        lista_1[0]

Out[3]: 4

In [4]: print(lista_1[3])
        print(lista_1[2])

        6
        3
```

El orden de los elementos se mantiene a menos que algo lo modifique, así que la sentencia lista_1[0] me va a regresar siempre un 4 (el primer elemento de la lista), a menos que la lista sea modificada. Es importante tener cuidado cuando accedemos a índices en una lista. Si pedimos un índice que no está ligado a ningún elemento (porque es un índice mayor al número de elementos que tenemos en la lista), vamos a obtener un error (algo que no es buena idea obtener, como podrán imaginarse):

```
In [7]: lista_1[5]

        ---------------------------------------------------------------------------
        IndexError                                Traceback (most recent call last)
        <ipython-input-7-f2e8ba98b710> in <module>
        ----> 1 lista_1[5]

        IndexError: list index out of range
```

Figure 4: IMAGEN 5

## APPEND

Esto nos lleva a nuestra siguiente pregunta: ¿cómo es que modificamos una lista? Vamos a aprender 2 métodos básicos para modificar listas en Python: append y pop. Append Es una función que agrega un elemento al final de la lista. Llamamos append como hemos llamado otras funciones (print o type), usando paréntesis y pasándole dentro del paréntesis el elemento que queremos agregar a la lista. La única diferencia es que esta función la llamamos desde el objeto de la lista, con la sintaxis variable_que_contiene_tu_lista.append(). Es decir, variable_que_contiene_tu_lista + . + append + (). Porque esta función la llamamos desde el objeto de la lista, la nombramos método en vez de lista. No importa si no entiendes por qué en ese momento, pero yo voy a llamarles métodos a partir de ahora para ser correctos. Veamos append en acción:

```
In [5]: lista_2 = [4, 7, 6, 2, 5]

        lista_2.append(10)

        lista_2

Out[5]: [4, 7, 6, 2, 5, 10]
```

Figure 5: IMAGEN 6

Una vez, agregada, podemos acceder a ella con su nuevo índice (el último de la lista):

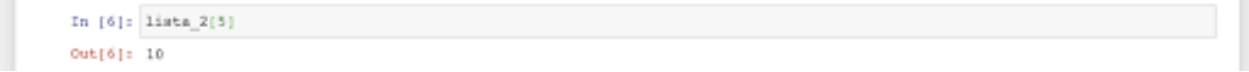


Figure 6: IMAGEN 7

### POP

Ya tenemos un método que agrega elementos a la lista. Ahora queremos quitar elementos de la lista, pop se utiliza justamente para eso. Si llamamos este método sin pasarle nada, pop remueve el último elemento de la lista:


Figure 7: IMAGEN 8

Podemos pasarle un índice a pop para que remueva el elemento que está en ese índice:

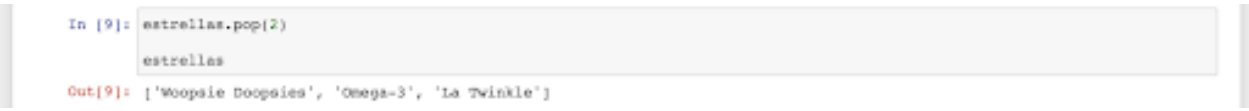


Figure 8: IMAGEN 9

Como podrás imaginar, al remover elementos en índices intermedios, todos los elementos posteriores a ese índice tienen ahora un índice menor. Más exactamente indice_anterior - 1. Esto es importante porque las listas siempre deben de tener un índice secuencial. Si hubiera huecos en nuestro índice, sería imposible saber qué índices acceder y cuáles no. Esto quiere decir que para acceder a “La Twinkle” tenemos que usar el índice 2, no el 3 como anteriormente:

## DICCIONARIOS

Los diccionarios se parecen bastante a los diccionarios léxicos. En un diccionario léxico, tienes pares de palabra-significado. si quieres saber el significado de una palabra, vas y buscas la palabra. Esa palabra está ligada a su significado. Por lo tanto, los diccionarios están organizados por pares. De igual manera, los diccionarios en Python están organizados en pares “llave-valor”. Cada llave tiene un valor asignado y para acceder al valor basta con pedir la llave. Una diferencia importante con los diccionarios léxicos es que los diccionarios en Python no están ordenados. Esto en realidad no importa demasiado, ya que buscar valores en un diccionario de Python es mucho más fácil y eficiente que en un diccionario léxico. ¿Y cómo es que creamos diccionarios? De esta manera:

Como ves, para crear diccionarios usamos llaves **{ }** en vez de corchetes. Después, escribimos el nombre de una llave y su respectivo valor separados por dos puntos (:). Cada par llave-valor está separado de los demás por comas. En este caso las llaves son strings, pero también podrían ser números:

Ahora, cómo accedemos a los valores que hemos guardado en nuestro diccionario. Es simple: usamos corchetes como con las listas, pero en vez de pasar un índice, pasamos el nombre de la llave:

Fácil, ¿no? Agregar datos Agregar datos a nuestros diccionarios es bastante sencillo. Para agregar un dato, se realiza una asignación como ésta:

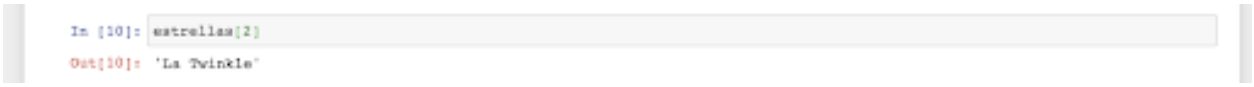


Figure 9: IMAGEN 10

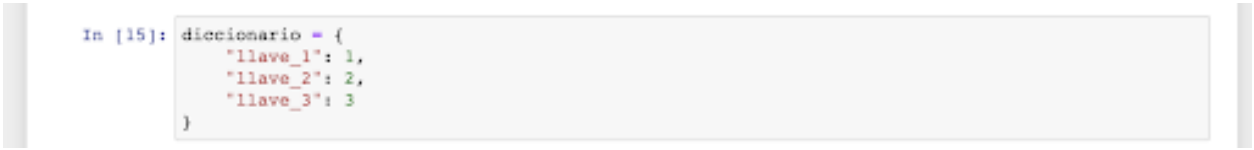


Figure 10: IMAGEN

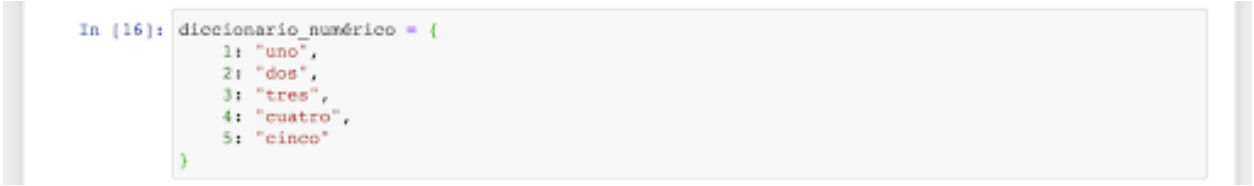


Figure 11: IMAGEN


Figure 12: IMAGEN

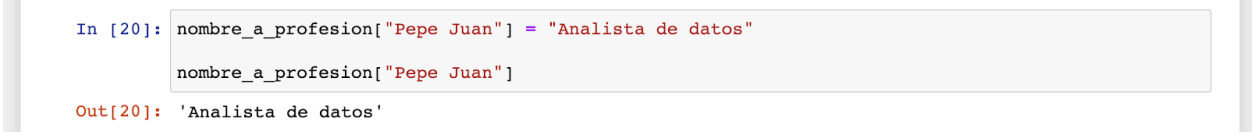


Figure 13: IMAGEN

Como ves, “selecciono” la llave que aún no existe y simplemente le paso el valor que le quiero asignar. Como se ve en la imagen, esa llave ahora existe en mi diccionario y puedo acceder a ella.

**Modificar datos**

También puedo modificar datos que ya se encuentran en mi diccionario. Para hacer esto, simplemente vuelo a asignar el nuevo valor a la llave que quiero modificar. Por ejemplo, digamos que Jenny acaba de cambiar de profesión. Podemos actualizar nuestro diccionario de la siguiente manera:

```
In [21]: nombre_a_profesion["Jenny"] = "Coach de equipo de futbol femenil"

         nombre_a_profesion["Jenny"]

Out[21]: 'Coach de equipo de futbol femenil'
```

Figure 14: IMAGEN

Los valores de los diccionarios también pueden ser listas u otros diccionarios (¡Inception!). Por ejemplo, mira esto:

```
In [22]: datos_compras = {
             "nombre": "Alberto Suarez",
             "productos_comprados": ["TV", "Chromebook", "VGA-USB-C"],
             "direccion_de_facturacion": {
                 "colonia": "Escandon",
                 "calle": "Mutualismo",
                 "numero": 44,
                 "cp": 11800
             }
         }
```

Figure 15: IMAGEN

En un caso como éste, también es posible modificar la lista y el diccionario que hay dentro. Es tan simple como pedir el valor y luego modificar ese valor (la lista o diccionario) con las técnicas que ya conocemos:

```
In [25]: datos_compras["productos_comprados"].append("Ipad Nano")

         datos_compras["direccion_de_facturacion"]["numero"] = 55

         pprint(datos_compras)

{'direccion_de_facturacion': {'calle': 'Mutualismo',
                              'colonia': 'Escandon',
                              'cp': 11800,
                              'numero': 55},
 'nombre': 'Alberto Suarez',
 'productos_comprados': ['TV',
                         'Chromebook',
                         'VGA-USB-C',
                         'Ipad Nano',
                         'Ipad Nano']}
```

Figure 16: IMAGEN

Lo siento estrella celeste “Justin Bieber”, siempre estarás en nuestros corazones.

Con esto terminamos la sección de estructuras de datos. Pasemos ahora a una de las herramientas más famosas y utilizadas en el mundo de la programación: las funciones.

## LAS FUNCIONES

Imaginemos que tenemos un código como éste (bueno, no hace falta imaginarlo, podemos escribirlo en nuestro JN):

```
In [27]: # Cantidad de alumnos estudiando en la UNAM en el ciclo escolar 2019-2020
         #
         # Fuente: http://www.estadistica.unam.mx/numeralia/
         #

         total_de_alumnos = 360883
         alumnos_en_posgrado = 30634
         alumnos_en_licenciatura = 217808
         alumnos_de_bachillerato = 111569
         alumnos_en_prope_de_musica = 872
```

Figure 17: IMAGEN

Queremos saber el porcentaje del total de alumnos que estudian en cada uno de los niveles distintos. Un primer acercamiento podría ser éste:

```
In [31]: alumnos_en_posgrado_porc = alumnos_en_posgrado * 100 / total_de_alumnos
         alumnos_en_licenciatura_porc = alumnos_en_licenciatura * 100 / total_de_alumnos
         alumnos_en_bachillerato_porc = alumnos_en_bachillerato * 100 / total_de_alumnos
         alumnos_en_prope_de_musica_porc = alumnos_en_prope_de_musica * 100 / total_de_alumnos
```

Figure 18: IMAGEN

No se ve tan mal, ¿verdad? Pues en realidad hay un problema más o menos grave: estamos repitiendo mucho código. Esa operación matemática para sacar el porcentaje, la regla de tres, ¡es exactamente igual siempre y la estamos escribiendo cuatro veces!

Si quiero sacar el porcentaje de otra estadística, tendría que volver a escribir la regla de tres. ¿Qué solución tengo? Crear una especie de "contenedor" donde tenga la lógica de mi operación, para que pueda repetir el proceso varias veces sin tener que volver a escribirlo. Justamente para eso sirven las funciones.

## SINTAXIS

Para declarar una función seguimos los siguintes pasos: 1. Empezamos con la palabra def 2. Elegimos un nombre para nuestra función (por favor nota que los nombres de las funciones también se escriben usando camel_case y siguen las mismas convenciones que los nombres de las variables) 3. Luego agregamos parentesis 4. Dentro de los parentesis agregamos los nombres de los parametros. Estos parámetros son datos que la función requiere para funcionar correctamente (podemos pensarlos como el ingrediente de nuestro proceso), 5. Agregamos al fiunal dos puntos y presionamos enter. RECUERDA INDENTACIONES

```
In [ ]: def sumar_dos_a_numero(numero):
            # Esto está adentro de la función

            # Esto también

        # Esto ya no
```

6. Escribir el proceso a seguir dentro de la indentación: 7. Retornar resultado

Finalmente quedandonos lo siguiente:

```
In [ ]: def sumar_dos_a_numero(numero):
            suma = numero + 2
            return suma
```

Por ejemplo tenemos la siguiente función:

```
In [35]: suma_de_5_mas_2 = sumar_dos_a_numero(5)
         suma_de_5_mas_2
Out[35]: 7
```

Figure 19: IMAGEN

¿Qué pasó aquí? Tengo una variable llamada suma_de_5_mas_2. A esta variable voy a asignarle el resultado de llamar mi función sumar_dos_a_numero con un 5 como argumento (se llaman parámetros dentro de la función pero argumentos a la hora de pasarlos... es confuso, lo sé). Dicho de otra manera: Llamo mi función sumar_dos_a_numero; le paso un 5 para que le sume 2; regreso el resultado de esa suma (7) y asigno ese resultado a una variable llamada suma_de_5_mas_2 que después puedo usar para lo que quiera (en este caso solamente estoy checando qué hay dentro de ella).

Lo interesante comienza cuando reutilizamos la función pasándole distintos argumentos:

```
In [36]: print(sumar_dos_a_numero(3))
         print(sumar_dos_a_numero(10))
         print(sumar_dos_a_numero(222))
         print(sumar_dos_a_numero(1036))

         5
         12
         224
         1038
```

Figure 20: IMAGEN

En este caso estoy imprimiendo los resultados de las llamadas a sumar_dos_a_numero sin asignarlos antes a una variable.

## FUNCIÓN PARA CUANTIFICAR PORCENTAJES

Regresemos entonces a nuestro problema original: calcular los porcentajes del total de alumnos de la UNAM que estudian en cada nivel.

Vamos a construir una función que realice esta operación:

```
In [37]: def calcular_porcentaje(muestra, total):
             porcentaje = muestra * 100 / total
             return porcentaje
```

Figure 21: IMAGEN

Estoy declarando esta función con dos parámetros: muestra y total, muestra es el subconjunto del total que quiero expresar en porcentajes, total es el total de la población de mi conjunto de datos. Puedo declarar funciones con un número indeterminado de parámetros, cuantos sea que necesite, pero en general se recomienda mantener el número de parámetros a lo mínimo necesario. Entre menos parámetros, mejor.

Ahora sí, vamos a utilizar nuestra función:

```
In [38]: total_de_alumnos = 360883
         alumnos_en_posgrado = 30634
         alumnos_en_licenciatura = 217888
         alumnos_en_bachillerato = 111569
         alumnos_en_prope_de_musica = 872

In [39]: alumnos_en_posgrado_porc = calcular_porcentaje(alumnos_en_posgrado, total_de_alumnos)
         alumnos_en_licenciatura_porc = calcular_porcentaje(alumnos_en_licenciatura, total_de_alumnos)
         alumnos_en_bachillerato_porc = calcular_porcentaje(alumnos_en_bachillerato,  total_de_alumnos)
         alumnos_en_prope_de_musica_porc = calcular_porcentaje(alumnos_en_prope_de_musica, total_de_alumnos)
```

Figure 22: IMAGEN

En este caso pareciera que no estamos ahorrando tanto trabajo, ¿no? La cantidad de código que estamos escribiendo es prácticamente la misma. Pero imagina que el proceso de sacar porcentajes fuera mucho más largo y complicado.

En ese caso tendría mucho más sentido. Además, tenemos otras ventajas que voy a enumerar a continuación.

## OTRAS VENTAJAS DEL USO DE FUNCIONES

- Nuestra función tiene un nombre (calcular porcentaje). Por lo tanto, si alguien está leyendo nuestro código y no sabe que la operación para obtener el porcentaje es muestra * 100 / total, puede enterarse de ello a través del nombre de nuestra función. Es algo muy bueno hacer nuestro código lo más comprensible posible, ¡nunca sabemos quién más va a tener que leerlo y entenderlo además de nosotros!
- En caso de que descubramos una operación más eficiente para calcular el porcentaje (en este caso es muy poco probable que suceda, pero puede suceder fácilmente con funciones más complejas), lo único que tenemos que hacer es ir a nuestra función y cambiar el código que hay dentro. Automáticamente, el resto de nuestro código que está utilizando esta función va a utilizar la función optimizada. Si no hubiéramos escrito una función, tendríamos que buscar todos los lugares donde realizamos esa operación para cambiarlos. Una regla de oro en la programación es: ¡Nunca repitas código! A la hora de corregir errores u optimizar tu código, vas a agradecer no repetir código.
- En general es una buena idea pensar en nuestros programas modularmente. Esto significa pensarlo en los procesos más simples posibles y esos procesos encapsularlos en funciones (u otras estructuras). Separar nuestro código en “cachitos” va a hacerlo mucho más fácil de entender, corregir y aumentar.

## CONTEXTO DE LAS VARIABLES

Para terminar este Prework, vamos a hablar de algo llamado contexto en Python. contexto se refiere al área en tu código donde cada variable puede ser accesada. No todas las variables en tu código pueden ser accesadas desde cualquier lugar. Vamos a ver:

```
In [42]: variable_global = "Global"
```

Figure 23: IMAGEN

Tenemos una variable llamada variable_global. Esta variable fue asignada en lo que se llama el contexto global. Podemos saber que es el contexto global porque no hay ni un solo nivel de indentación antes del nombre de la variable. Al estar en el contexto global, esta variable puede ser accedida desde cualquier lugar, en la misma celda en la que fue asignada o en alguna otra celda distinta:

Vemos ahora qué pasa con una variable declarada dentro de una sentencia if:

Como puedes ver, podemos acceder a la variable respuesta desde cualquier lugar. Eso quiere decir que en Python una sentencia if no cambia el contexto actual. Seguimos estando en el contexto desde el cual se declaró la sentencia if. Como esta sentencia se declaró en el contexto global, las variables que asignemos dentro del bloque de la sentencia seguirán teniendo contexto global.

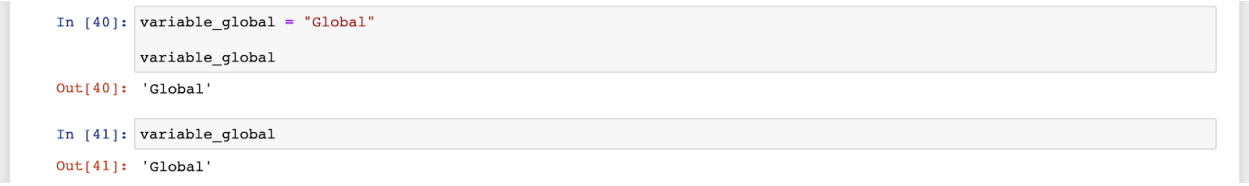


Figure 24: IMAGEN


Figure 25: IMAGEN

Pero veamos ahora qué pasa con las variables declaradas dentro de una función:


Figure 26: IMAGEN

Ok, declaramos la función, corrimos la celda y no hemos podido acceder a ninguna de las variables (ni el parámetro número ni la variable suma asignada dentro de la función). ¿Será porque no hemos llamado la función con algún valor como argumento? Veamos:

No, no tenemos acceso. ¿Qué está pasando? Lo que está pasando es que las funciones en Python sí cambian el contexto de las variables que "viven" dentro de la función.

Todas las variables que viven dentro de la función, tanto parámetros como variables asignadas, sólo pueden ser accesadas desde dentro de la función. Su contexto dejó de ser global y se convierte en el contexto de la función que las contiene.

La única manera de acceder a los datos que hemos generado dentro de la función es a través del return:

Los valores regresados en el return, "traspasan" contextos. Son como viajeros intergalácticos que pasan de una dimensión a otra a través de un hoyo negro. Igualito...


Figure 27: IMAGEN


Figure 28: IMAGEN

# WORK. ESTRUCTURAS DE DATOS

## OBJETIVOS

1. Comprender la utilidad de las estructuras de datos para organizar información dentro de nuestros programas.
2. Utilizar listas y diccionarios y entender las principales diferencias entre ellos.
3. Declarar funciones y utilizarlas perfectamente

## CONTENIDO

### LISTAS

Las listas son uno de los tipos de estructuras de datos que podemos utilizar en Python. Una estructura de datos nos sirve para organizar datos y para optimizar el acceso, procesamiento y uso de éstos.

Las listas son secuencias ordenadas de datos y se ven así:

lista_de_numeros = [1, 4, 7, 5, 3, 4, 6]

Vayamos a un primer ejemplo para entender su funcionamiento.

# RETO 1. CREANDO LISTAS Y ACCEDIENDO A ELLAS

**a) Definiendo una lista** Debajo de esta celda hay un nombre de una variable que aún no ha sido asignada:

```
mi_informacion =
```

Ahora, tenemos un print que imprime una string formateada para mostrar cierta información acerca de ti (Las triples comillas "' nos sirven para generar strings de varias líneas):

print(f"' ¡Hola! Mi nombre es {mi_informacion[0]}. Todos me dicen {mi_informacion[1]}. Mi comida favorita es {mi_informacion[2]}. Y la comida que más detesto es {mi_informacion[3]}. Mi trabajo ideal sería {mi_informacion[4]}. ¡Gracias, chau! "')

Para que esta string funcione, asigna una lista a la variable mi_informacion con toda la información necesaria para imprimir tu micro-bio debajo de la celda.

SOLUCIÓN:

```
mi_informacion=["VICTOR TERRON","TERRON","CALDO DE POLLO","TODO TIPO DE CUEROS","EMPRESARIO"]
print(f'''
¡Hola! Mi nombre es {mi_informacion[0]}. Todos me dicen {mi_informacion[1]}.
Mi comida favorita es {mi_informacion[2]}. Y la comida que más detesto es {mi_informacion[3]}.
Mi trabajo ideal sería {mi_informacion[4]}.
¡Gracias, chau!
''')
```

```
¡Hola! Mi nombre es VICTOR TERRON. Todos me dicen TERRON.
Mi comida favorita es CALDO DE POLLO. Y la comida que más detesto es TODO TIPO DE CUEROS.
Mi trabajo ideal sería EMPRESARIO.
¡Gracias, chau!
```

### b) Una lista construida con variables

Debajo de esta celda vemos los nombres de varias variables y una lista que las contiene.

info_0 = info_1 = info_2 = info_3 = info_4 =

info_faltante[info_0, info_1, info_2, info_3, info_4]

Ahora, tenemos un print que imprime una pequeña historia utilizando la lista info_faltante que contiene a las demás variables:

print(f"' Algún día los {info_faltante[0]} lograrán su objetivo. Su objetivo de {info_faltante[1]}. Ese día, los humanos {info_faltante[2]} y tendrán que {info_faltante[3]}. Por esa razón yo todos los días {info_faltante[4]}. "')

Tu reto será asignar las variables info_x con la información que desees para crear una historia a tu gusto. Cada variable puede contener una string que tenga varias palabras (frases, pues). Si te dan ganas, ¡comparte la historia con tus compañeros!

```
info_0 = "ALIENS"
info_1 = "TENER UN PERRO"
info_2 = "DORMIRÁN"
info_3 = "REGALAR PERROS"
info_4 = "CUIDO A MI CACHORRA"

info_faltante=[info_0, info_1, info_2, info_3, info_4]

print(f'''
Algún día los {info_faltante[0]} lograrán su objetivo. Su objetivo de {info_faltante[1]}.
Ese día, los humanos {info_faltante[2]} y tendrán que {info_faltante[3]}.
Por esa razón yo todos los días {info_faltante[4]}.
''')
```

```
Algún día los ALIENS lograrán su objetivo. Su objetivo de TENER UN PERRO.
Ese día, los humanos DORMIRÁN y tendrán que REGALAR PERROS.
Por esa razón yo todos los días CUIDO A MI CACHORRA.
```

## OPERADOR DE INDEXACIÓN RETO 2

Debajo de esta celda hay una lista con números dentro:

respuestas = [0.58, 9, 2, 3, 37, 5, 75, 4]

Ahora, tenemos un print con una string interpolada. Como puedes ver todas las interpolaciones están vacías:

print(f"' 1. Los humanos tenemos {} ojos en la cara. 2. Un humano adulto tiene {} dientes dentro de su boca. 3. Un feto tarda {} meses en gestarse antes de nacer. 4. La expectativa de vida en México es de alrededor de {} años. 5. Las horas de sueño al día recomendadas para adultos jóvenes son entre {} y {}. 6. El récord actual de velocidad en 100 metros (09/05/2020) fue establecido por Usain Bolt y es de {} "')

```
respuestas = [0.58, 9, 2, 3, 37, 5, 75, 4]

print(f'''
1. Los humanos tenemos {respuestas[2]} ojos en la cara.
2. Un humano adulto tiene {respuestas[4]} dientes dentro de su boca.
3. Un feto tarda {respuestas[1]} meses en gestarse antes de nacer.
4. La expectativa de vida en México es de alrededor de {respuestas[6]} años.
5. Las horas de sueño al día recomendadas para adultos jóvenes son entre {respuestas[7]+respuestas[2]} y
6. El récord actual de velocidad en 100 metros (09/05/2020) fue establecido por Usain Bolt y es de {resp
''')
```

```
1. Los humanos tenemos 2 ojos en la cara.
2. Un humano adulto tiene 37 dientes dentro de su boca.
3. Un feto tarda 9 meses en gestarse antes de nacer.
4. La expectativa de vida en México es de alrededor de 75 años.
5. Las horas de sueño al día recomendadas para adultos jóvenes son entre 6 y 5.
6. El récord actual de velocidad en 100 metros (09/05/2020) fue establecido por Usain Bolt y es de 21.45
```

Se puede acceder con indices negativos.

# RETO 2. MODIFICACIÓN DE LISTAS

## OBJETIVO

1. Modificar listas
2. Aplicar *append* y *pop*

## DESARROLLO

### Modificar para eliminar diferencias

Debajo tienes dos listas definidas:

```
lista_1=[3.4, 0.7, 99.9, 5.41, 6.23, 7.9]
```

```
lista_2 = [3.4, 63.4, 0.7, 6.46, 99.9, 2.2, 5.41]
```

Ahora, tienes una sentencia if que utiliza una comparación entre las dos listas:

```
if lista_1 == lista_2: print("Tú y yo somos uno mismo")
```

Donde dice: tu código va aquí, agrega modificaciones a la lista_2 para lograr que ambas listas sean idénticas y el bello mensaje "Tú y yo somos uno mismo" se imprima. Puedes usar tanto **pop** como **append**, pero sólo puedes modificar la **lista_2**.

```
lista_1 = [3.4, 0.7, 99.9, 5.41, 6.23, 7.9]
lista_2 = [3.4, 63.4, 0.7, 6.46, 99.9, 2.2, 5.41]
lista_2.pop(1)
```

```
63.4
```

```
print(lista_2)
```

```
[3.4, 0.7, 6.46, 99.9, 2.2, 5.41]
```

```
lista_2.pop(2)
```

```
6.46
```

```
print(lista_2)
```

```
[3.4, 0.7, 99.9, 2.2, 5.41]
```

```
lista_2.pop(3)
```

```
2.2
```

```
print(lista_2)
```

```
[3.4, 0.7, 99.9, 5.41]
```

```
lista_2.append(6.23)
lista_2.append(7.9)
if lista_1 == lista_2:
    print("Tú y yo somos uno mismo")
```

```
Tú y yo somos uno mismo
```

```
fragmento_fragmentado = ['Apenas', 'él', 'le', 'amalaba', 'hidrolizado' 'el', 'noema,',
                         'a', 'ella', 'se', 'le', 'agolpaba', 'el', 'clémiso', 'súbito',
                         'y', 'caían', 'en', 'fermales', 'hidromurias,', 'en', 'salvajes', 'ambonios,',
                         'en', 'sustalos', 'distales', 'exasperantes.',
                         'Cada', 'vez', 'que', 'él', 'procuraba', 'relamar', 'las', 'incopelusas,']

tu_nombre = "TERRON"  # Tu nombre va aquí

print(f'==Colaboración póstuma de Córtazar y {tu_nombre}==\n')
```

```
==Colaboración póstuma de Córtazar y TERRON==
```

```
print(f'{" ".join(fragmento_fragmentado)}')
```

```
Apenas él le amalaba hidrolizadoel noema, a ella se le agolpaba el clémiso súbito y caían en fermales hi
```

# RETO 3. CREANDO Y ACCESANDO DICCIONARIOS

## OBJETIVOS

- ENTENDER CÓMO CREAR DICCIONARIOS

## DESARROLLO

Debajo se ha definido un diccionario. Éste diccionario se encuentra incompleto, aunque no lo parezca:

```
ventas_mensuales = {
    "fecha_de_corte": "01/05/2020",
    "unidad": "Tlaxcala",
    "ventas_pasteleria":"10000",
    "ventas_panaderia":"1000",
    "ganancias_pasteleria":"999",
    "ganancias_panaderia":"99",
    "gastos_mensuales_totales":"100",
    "analista":"Victor Terron"
}
```

Debajo tenemos algunos procedimientos que han sido realizados utilizando este diccionario. Como puedes ver, algunos de los accesos que se están haciendo van a fallar porque el diccionario ventas_mensuales no está completo:

```
ventas_totales_de_insumos= int(ventas_mensuales["ventas_pasteleria"]) + int(ventas_mensuales["ventas_pa
ganancias_totales = int(ventas_mensuales["ganancias_pasteleria"]) + int(ventas_mensuales["ganancias_pana
ganancias_netas = int(ganancias_totales) - int(ventas_mensuales["gastos_mensuales_totales"])

print(f'==Resumen de ventas mensuales de la unidad {ventas_mensuales["unidad"]}==/n')
```

```
==Resumen de ventas mensuales de la unidad Tlaxcala==/n
```

```
print(f'Fecha de corte: {ventas_mensuales["fecha_de_corte"]}/n')
```

```
Fecha de corte: 01/05/2020/n
```

```
print(f'  - Ventas totales de insumos: {ventas_totales_de_insumos}')
```

```
  - Ventas totales de insumos: 11000
```

```
print(f'  - Ganancias totales: {ganancias_totales}')
```

```
  - Ganancias totales: 1098
```

```
print(f'  - Ganancias netas: {ganancias_netas}')
```

```
  - Ganancias netas: 998
```

```
print(f'\n')
```

```
print(f'(Información recabada por: {ventas_mensuales["analista"]})')
```

```
(Información recabada por: Victor Terron)
```

# RETO 4. MODIFICANDO DICCIONARIOS

## OBJETIVOS

- Practicar agregación de datos, modificación de datos y eliminación de diccionarios

## DESARROLLO

Debajo tienes un diccionario que contiene algo de información sobre una persona:

```
from pprint import pprint
registro = {
    "id": "23f-58j-kju7-54re",
    "nombre": "Alberto",
    "apellido_materno": "Gutierrez",
    "apellido_paterno": "Sosa",
    "profesion": "Contador",
    "ultimo_nivel_de_estudios": "Maestría",
    "lugar_de_estudios": "UAM",
    "numero_de_cuenta": "25367890",
    "nip_de_cajero": "142"
}
```

Ahora, tenemos una serie de prints que imprimen esta información en forma de tabla:

```
print(f'Registro con id: {registro["id"]}\n')
print(f'----------------------------------------\n')
print(f'{("Nombre"):25} | {registro["nombre"]:25}')
print(f'{("Apellido Materno"):25} | {registro["apellido_materno"]:25}')
print(f'{("Apellido Paterno"):25} | {registro["apellido_paterno"]:25}')
print(f'{("Profesión"):25} | {registro["profesion"]:25}')
print(f'{("Último nivel de estudios"):25} | {registro["ultimo_nivel_de_estudios"]:25}')
print(f'{("Lugar de estudios"):25} | {registro["lugar_de_estudios"]:25}')
print(f'{("Fecha de nacimiento"):25} | {registro["fecha_de_nacimiento"]:25}')
print(f'{("Lugar de nacimiento"):25} | {registro["lugar_de_nacimiento"]:25}')
```

La actividad tienen 3 partes:

1. Usando la técnica para modificar valores en un diccionario, cambia la información del diccionario para que sea la tuya.
2. Usando la técnica para agregar datos al diccionario agrega las llaves que estén siendo accesadas en el print pero que no han sido agregadas al diccionario.
3. Usando la técnica para eliminar datos del diccionario, elimina los datos sensibles que no quieras que estén incluidos en el diccionario.

Recuerda sólo utilizar métodos y operadores para realizar el ejercicio. **No reescribas el diccionario directamente**.

## SOLUCIÓN TERRON

```
from pprint import pprint
registro={
    "id": "TEMV991023HTLRCC02",
    "nombre": "Victor Miguel",
    "apellido_materno": "Macias",
    "apellido_paterno": "Terron",
    "profesion": "Ingeniero",
    "ultimo_nivel_de_estudios": "Licenciatura",
    "lugar_de_estudios": "UTT",
    "numero_de_cuenta": "569856",
    "nip_de_cajero": "9865",
}
registropersonal=registro.copy()
print(f'Registro con id: {registro["id"]}\n')
```

```
Registro con id: TEMV991023HTLRCC02
```

```
print(f'----------------------------------------\n')
```

```
----------------------------------------
```

```
print(f'{("Nombre"):25} | {registropersonal["nombre"]:25}')
```

```
Nombre                    | Victor Miguel
```

```
print(f'{("Apellido Materno"):25} | {registropersonal["apellido_materno"]:25}')
```

```
Apellido Materno          | Macias
```

```
print(f'{("Apellido Paterno"):25} | {registropersonal["apellido_paterno"]:25}')
```

```
Apellido Paterno          | Terron
```

```
print(f'{("Profesión"):25} | {registropersonal["profesion"]:25}')
```

```
Profesión                 | Ingeniero
```

```
print(f'{("Último nivel de estudios"):25} | {registro["ultimo_nivel_de_estudios"]:25}')
```

```
Último nivel de estudios  | Licenciatura
```

```
print(f'{("Lugar de estudios"):25} | {registro["lugar_de_estudios"]:25}')
```

```
Lugar de estudios         | UTT
```

```
registro["fecha_de_nacimiento"]="23/10/1999"
registro["lugar_de_nacimiento"]="Tlaxcala"
print(f'{("Fecha de nacimiento"):25} | {registro["fecha_de_nacimiento"]:25}')
```

```
Fecha de nacimiento       | 23/10/1999
```

```
print(f'{("Lugar de nacimiento"):25} | {registro["lugar_de_nacimiento"]:25}')
```

```
Lugar de nacimiento       | Tlaxcala
```

```
registropersonal.pop("numero_de_cuenta")
```

```
'569856'
```

```
registropersonal.pop("nip_de_cajero")
```

```
'9865'
```

```
pprint(registropersonal)
```

```
{'apellido_materno': 'Macias',
 'apellido_paterno': 'Terron',
 'id': 'TEMV991023HTLRCC02',
 'lugar_de_estudios': 'UTT',
 'nombre': 'Victor Miguel',
 'profesion': 'Ingeniero',
 'ultimo_nivel_de_estudios': 'Licenciatura'}
```

# RETO 5.FUNCIONES

## OBJETIVO

1. Practicar la declaración de funciones
2. Practicar la definición de parámetros y su uso dentro de la función
3. Practicar el uso de los valores retornados por una función
4. Practicar el uso de funciones para evitar la repetición de código

## DESARROLLO

### FUNCIÓN numero_es_par

Debajo tienes una funcion incompleta:

```
def numero_es_par(numero):

    # Tu código va aquí
    # ...
    # ...

    return
```

Dicha función debería de tomar un parámetro numero, checar si el número es par, regresar True si el número es par y regresar False si el número no es par.

Completa la función para que el código de abajo (que realiza tests de la función) regrese todos los resultados esperados.

**Pista:** Si no conoces el operador módulo (%), pídele al experto que explique su funcionamiento.

**PD:** Si no entiendes la función test_funcion no te preocupes, por el momento sólo está ahí para probar tu código y que sepas si tu función funciona correctamente.

```
def test_funcion(funcion_a_probar,
                 num_de_errores,
                 contador,
                 parametros=[],
                 resultado_esperado=None):
    resultado_test = funcion_a_probar(*parametros)
    print(f'''Test {contador}:
            Resultado esperado es `{resultado_esperado}`,
            obtuvimos `{resultado_test}`''')
    if resultado_test != resultado_esperado:
        num_de_errores += 1

    return num_de_errores
```

## SOLUCION PARTE 1

```
bandera=False
def numero_es_par(numero):
    if(numero%2==0):
      bandera=True
    else:
      bandera=False

    return bandera
```

Haciendo la prueba de la función:

```
numero_es_par(5)
```

```
False
```

```
numero_es_par(4)
```

```
True
```

```
print("== Tests numero_es_par==\n")
```

```
## == Tests numero_es_par==
```

```
contador_de_tests = 0
errores = 0

contador_de_tests += 1
errores = test_funcion(numero_es_par,
                       errores,
                       contador_de_tests,
                       parametros=[2],
                       resultado_esperado=True)
```

```
## Test 1:
##             Resultado esperado es ‘True‘,
##             obtuvimos ‘True‘
```

```
contador_de_tests += 1
errores = test_funcion(numero_es_par,
                       errores,
                       contador_de_tests,
                       parametros=[3],
                       resultado_esperado=False)
```

```
## Test 2:
##             Resultado esperado es ‘False‘,
##             obtuvimos ‘False‘
```

```
contador_de_tests += 1
errores = test_funcion(numero_es_par,
                       errores,
                       contador_de_tests,
                       parametros=[0],
                       resultado_esperado=True)
```

```
## Test 3:
##             Resultado esperado es ‘True‘,
##             obtuvimos ‘True‘
```

```
contador_de_tests += 1
errores = test_funcion(numero_es_par,
                       errores,
                       contador_de_tests,
                       parametros=[127],
                       resultado_esperado=False)
```

```
## Test 4:
##             Resultado esperado es ‘False‘,
##             obtuvimos ‘False‘
```

```
contador_de_tests += 1
errores = test_funcion(numero_es_par,
                       errores,
                       contador_de_tests,
                       parametros=[-88],
                       resultado_esperado=True)
```

```
## Test 5:
##             Resultado esperado es ‘True‘,
##             obtuvimos ‘True‘
```

```
contador_de_tests += 1
errores = test_funcion(numero_es_par,
                       errores,
                       contador_de_tests,
                       parametros=[-1349],
                       resultado_esperado=False)
```

```
## Test 6:
##             Resultado esperado es ‘False‘,
##             obtuvimos ‘False‘
```

```
print(f'\nErrores encontrados: {errores}')
```

```
##
## Errores encontrados: 0
```

## PARTE 2. REUTILIZACIÓN DEL CÓDIGO

Debajo tenemos algo de código.

```
resultado_1 = 34 * 100 / 100
print(f'34 es el {resultado_1}% del número 100\n')

resultado_2 = 57 * 100 / 127
print(f'57 es el {resultado_2}% del número 127\n')

resultado_3 = 12 * 100 / 228
print(f'12 es el {resultado_3}% del número 228\n')
```

```
resultado_4 = 87 * 100 / 90
print(f'87 es el {resultado_4}% del número 90\n')

resultado_5 = 1 * 100 / 999
print(f'1 es el {resultado_5}% del número 999\n')

resultado_6 = 66 * 100 / 66
print(f'66 es el {resultado_6}% del número 66\n')
```

Este código funciona correctamente, pero como puedes ver, estamos escribiendo el mismo código una y otra vez. En la celda debajo, escribe una función que realice la operación matemática que estamos realizando arriba, para que podamos reusarla múltiples veces para obtener los resultados que queremos. Llena también los prints de manera que podamos leer el resultado en un formato comprensible.

Reto extra: Si quieres un reto extra, escribe también una función que genere las strings que vamos a pasar a los prints, para no tener que escribirlas desde 0 cada vez.

```
def funcion(especimen,completo):
  return especimen*100/completo

def stringser(especimen,completo):
  porcentaje=especimen*100/completo
  return f'{especimen} es el {porcentaje}% del número total de: {completo}'




resultado_1 = 34 * 100 / 100
print(f'34 es el {resultado_1}% del número 100\n')
```

```
## 34 es el 34.0% del número 100
```

```
resultado_2 = 57 * 100 / 127
print(f'57 es el {resultado_2}% del número 127\n')
```

```
## 57 es el 44.881889763779526% del número 127
```

```
resultado_3 = 12 * 100 / 228
print(f'12 es el {resultado_3}% del número 228\n')
```

```
## 12 es el 5.2631578947368425% del número 228
```

```
resultado_4 = 87 * 100 / 90
print(f'87 es el {resultado_4}% del número 90\n')
```

```
## 87 es el 96.66666666666667% del número 90
```

```
resultado_5 = 1 * 100 / 999
print(f'1 es el {resultado_5}% del número 999\n')
```

```
## 1 es el 0.1001001001001001% del número 999
```

```
resultado_6 = 66 * 100 / 66
print(f'66 es el {resultado_6}% del número 66\n')
```

```
## 66 es el 100.0% del número 66
```

```
funcion(10,100)
```

```
## 10.0
```

```
stringser(10,100)
```

```
## '10 es el 10.0% del número total de: 100'
```

## PARTE 3. FUNCION acceso_autorizado

Debajo tenemos un conjunto de datos que tiene información de varios usuarios de una plataforma web. Este diccionario relaciona usernames con un rol y (a veces) con un nip de acceso:

```
usuarios = {
    "manolito_garcia": {
        "rol": "admin"
    },
    "sebas_macaco_23": {
        "rol": "editor",
        "nip_de_acceso": 3594
    },
    "la_susanita_maestra": {
        "rol": "admin"
    },
    "pepe_le_pu_88": {
        "rol": "lector"
    },
    "jonny_bravo_estuvo_aqui": {
        "rol": "editor",
        "nip_de_acceso": 9730
    },
    "alfonso_torres_69": {
        "rol": "editor",
        "nip_de_acceso": 2849
    },
    "jocosita_99": {
        "rol": "lector"
    }
}
```

Nuestra plataforma tiene 3 roles:

1. Admin: estos pueden editar información sin necesitar un nip de acceso.
2. Editor: pueden editar información sólo si escriben correctamenta su nip de acceso.
3. Lector: no pueden editar, sólo ver la información, no necesitan nip de acceso.

En la celda debajo, crea una función llamada nivel_de_acceso_para_username que reciba 3 parámetros:

1. base_de_datos: que será siempre nuestro diccionario usuarios.
2. username: El username del usuario que está solicitando acceso.
3. nip_de_acceso: El nip de acceso, que puede ser None en el caso de que el usuario no tenga uno (o que haya olvidado escribirlo a la hora de pedir acceso.

Con estos 3 parámetros, nuestra función tiene que regresar uno de los códigos de acceso siguientes:

- 0: Acceso denegado (esto sucede si el rol es editor y el nip_de_acceso es incorrecto).
- 1: Modo edición autorizada (esto sucede si el rol es admin o si el rol es editor y el nip_de_acesso es correcto).
- 2: Modo lectura autorizada (esto sucede si el rol es lector).

Después, corre los tests para asegurarte de que tu función es correcta.

Tip: Recuerda que puedes "anidar" sentencias if dentro de otras sentencias if.

Reto extra: Agrega un chequeo que regrese 0 si el username que recibió tu función no existe en la base de datos.

```
def nivel_de_acceso_para_username(base_de_datos,username,nip_de_acceso):
  #RETO EXTRA
  if username not in base_de_datos:
    return 0
  #RETO NORMAL
  if(base_de_datos[username]["rol"]=="admin"):
    return 1
  elif(base_de_datos[username]["rol"]=="lector"):
    return 2
  else:
    if base_de_datos[username]["nip_de_acceso"]==nip_de_acceso:
      return 1
    else:
      return 0
```

Realizando las pruebas tenemos lo siguiente:

```
print("== Tests nivel_de_acceso_para_username==\n")
```

```
## == Tests nivel_de_acceso_para_username==
```

```
contador_de_tests = 0
errores = 0

contador_de_tests += 1
errores = test_funcion(nivel_de_acceso_para_username, errores, contador_de_tests,
                       parametros=[usuarios, "manolito_garcia", None], resultado_esperado=1)
```

```
## Test 1:
##             Resultado esperado es ‘1‘,
##             obtuvimos ‘1‘
```

```
contador_de_tests += 1
errores = test_funcion(nivel_de_acceso_para_username, errores, contador_de_tests,
                       parametros=[usuarios, "sebas_macaco_23", 3594], resultado_esperado=1)
```

```
## Test 2:
##             Resultado esperado es `1`,
##             obtuvimos `1`
```

```
contador_de_tests += 1
errores = test_funcion(nivel_de_acceso_para_username, errores, contador_de_tests,
                       parametros=[usuarios, "jonny_bravo_estuvo_aqui", 9999], resultado_esperado=0)
```

```
## Test 3:
##             Resultado esperado es `0`,
##             obtuvimos `0`
```

```
contador_de_tests += 1
errores = test_funcion(nivel_de_acceso_para_username, errores, contador_de_tests,
                       parametros=[usuarios, "pepe_le_pu_88", None], resultado_esperado=2)
```

```
## Test 4:
##             Resultado esperado es `2`,
##             obtuvimos `2`
```

```
contador_de_tests += 1
errores = test_funcion(nivel_de_acceso_para_username, errores, contador_de_tests,
                       parametros=[usuarios, "alfonso_torres_69", None], resultado_esperado=0)

# Corre el siguiente código sólo si decidiste realizar el reto extra
```

```
## Test 5:
##             Resultado esperado es `0`,
##             obtuvimos `0`
```

```
contador_de_tests += 1
errores = test_funcion(nivel_de_acceso_para_username, errores, contador_de_tests,
                       parametros=[usuarios, 'los_yeah_yeahs_97', 1345], resultado_esperado=0)
```

```
## Test 6:
##             Resultado esperado es `0`,
##             obtuvimos `0`
```

```
print(f'\nErrores encontrados: {errores}')
```

```
##
## Errores encontrados: 0
```