Preguntas teóricas

1. *¿Cuál es la diferencia entre una lista y una tupla en Python?*

Una LISTA es **mutable; su sintaxis es mediante []**. Para Python una lista es una colección ordenada mutable, pero no tiene un control de cuántos elementos hay en ella, los detalles, o cómo se acceden. No tienen un tamaño fijo, y los elementos no se almacenan en un bloque de memoria contiguo. Las formas de acceso a ella son normalmente agregando un elemento al final (**append()**), o quitando el elemento final (**pop()**), o un blucle "**for**" integrado que itera de principio a fin.

También puedes eliminar elementos con **remove() o** con la palabra clave **del**

```
mi_lista = [1, 2, 3]
mi_lista.remove(1)    #se refiere al elemento, no a la posición
del mi_lista[0]       #se refiere a la posición
print(mi_lista)       #en ambos casos, devolvería [2, 3]
```

Hay casos en los que no quieres cambiar la variable en una lista. Esto se haría mediante el método **sorted().** Por ejemplo, si la necesitamos también del revés:

```
nueva_lista = sorted(mi_lista, reverse=True)
```

La TUPLA, cuyos elementos van entre **paréntesis (),** es **inmutable**. El hecho de no poder cambiar elementos también conlleva también que no puedes cambiarlos de orden, ya que, si lo hicieras, el valor que correspondería a otro índice. Ejemplo:

```
resp_ventas = ("Martínez", "Agudo", "Carmen")
print(resp_ventas.sort())  #----> nos da ERROR,
```

Da error porque al ordenar alfabéticamente una tupla, cambiaría el orden de los elementos y si funcionara, el resultado sería que en la pos. **[0]** ya no estaría el primer apellido "**Martínez**", sino que figuraría el segundo apellido ("**Agudo**"), y por definición, ya no sería una tupla.

SÍ que se podrían realizar CAMBIOS mediante un proceso más largo:

```
resp_ventas = list(resp_ventas)          #1 – convertir a LISTA
resp_ventas[2] = "Carmen María"
resp_ventas = tuple(resp_ventas)         #2 – convertir a TUPLA
```

Como vemos, hacer todo este proceso debería ser ocasional, porque en caso de necesitemos cambios frecuentes deberíamos preguntarnos por qué en un principio, elegimos usar una TUPLA en vez de una LISTA. Así que, en función de lo que nos convenga y se ajuste a nuestro caso, elegiremos la opción más adecuada.

2. ¿Cuál es el orden de las operaciones?

El orden en que se resuelven las operaciones en Python se puede identificar con las siglas **PEMDAS** (**P**aréntesis **(+)**, **E**xponentes**(**)**, **M**ultiplicación**(*)**, **D**ivisión**(/)**, **A**dición o suma**(+)**, **S**ustracción o resta**(-)**).

Ante la duda siempre podemos usar PARÉNTESIS, ya que todo lo que está dentro de un paréntesis se evaluará conjuntamente, pero es importante tener en cuenta las prioridades. Ejemplo:

```
total = 128 / 2 ** 5 - (1 + 2) * 4
    #   Primero el paréntesis (1+2) = 3,        quedaría  128 / 2 ** 5 - 3 * 4
    #   Seguido de exponentes 2**5 = 32,        quedaría  128 / 32 - 3 * 4
    #   Ahora la multiplicación 3*4 = 12,       quedaría  128 / 32 - 12
    #   Ahora la división 128/32=4,             quedaría  4 - 12          ---→ = -8

print(total)  #devolverá -8.0,  porque lo convierte en flotante al haber división)
```

3. ¿Qué es un diccionario Python?

Un **diccionario** en Python es una colección de elementos, donde cada uno tiene una llave *KEY* (que es única, y un valor *VALUE* (que puede ser cualquier cosa) separados por dos puntos (*KEY: VALUE*). Una clave puede ser y puede ser un carácter, una cadena de caracteres, enteros o booleanos o reales, y tuplas). Son iterables y mutables igual que las LISTAS, pero no son secuenciales (las LISTAS sí), es decir, no tiene sentido decir que un elemento va antes que otro, por lo que no se pueden realizar cortes o *SLICES/RANGES*. Se emplean mucho en bases de datos de sitios Web, aplicaciones de móviles y Machine Learning o aprendizaje automático. Los diccionarios se crean con **llaves {}.** Ejemplo:

```
especias_mundo = {"Tex-mex": 2.45, "India": 3.05, "Perú": 2.90, "Colombia ": 3.03}
```

Cuando se hace una consulta a un diccionario, se crea lo que se llama "*thread*" o "hilo". Para realizar consultas a diccionarios se suelen utilizar los siguientes métodos que retornan *VIEW OBJECTS* u objetos vista. Devuelven lo que se llama DICCIONARIO VISTA:

```
                                    #DEVUELVE:
#especias_mundo.keys()     #  dict_keys(['Tex-mex', 'India', 'Perú', 'Colombia '])
#especias_mundo.values()   #  dict_values([2.45, 3.05, 2.9, 3.03])
#especias_mundo.items()    #  dict_items([('Tex-mex', 2.45), ('India', 3.05), ('Perú', 2.9), ('Colombia ', 3.03)])
```

Como vemos, el **item()** retorna una LISTA de TUPLAS. Hay algunas funciones que podemos aplicar a una TUPLA, como **len(**dictview**)**, que en nuestro caso nos daría 4 elementos. También se puede manipular y convirtiéndolo a LISTA → **list(**especias del mundo**)** y hacer consultas de los elementos con los [índice]s con los que podemos sacar los elementos de listas.

Un DICCIONARIO VISTA es **dinámico**, lo que significa que si un usuario intenta cambiar los valores de una lista al mismo tiempo que otro proceso -por ejemplo un cambio o consulta- por parte de otro usuario, pues se accederá siempre AL ÚLTIMO CAMBIO, ya que cambia A TIEMPO REAL, es decir los cambios están disponibles en el acto.

Si queremos asegurarnos de tener una lista con los valores iniciales, es decir, los que teníamos en mente cuando decidimos obtener esa lista de valores (libre de otros cambios que se estén produciendo), podemos crear una COPIA* de la lista de esta forma:

```
mis_especias = list(especias_mundo.copy().values())    #un 'print' devolvería [2.45, 3.05, 2.9, 3.03]
```

*(*este tipo de COPIA se llama "SHALLOW COPY" o copia superficial, ya que no realiza copia de otras estructuras anidadas, como listas u otros diccionarios; para esto último existe la "DEEP COPY" o copia profunda).*

Por otra parte, es especialmente interesante el método **get().** Las funciones **get()** y **values()** se aplican a diccionarios, pero NO a LISTAS. La estructura **get()** es:

```
get(<key>[,<default>])
```

Nos permite consultar el valor para una clave determinada. Si proporcionamos el SEGUNDO PARÁMETRO, que es OPCIONAL, nos devolvería el valor que nosotros pongamos en caso de la clave que hemos introducido estuviera mal escrita, o no existiera, y así nos puede dar un aviso en consecuencia:

```
print((especias_mundo.get("INdia", "No encontrado")))    #  devuelve "No encontrado"
```

4. *¿Cuál es la diferencia entre el método ordenado y la función de ordenación?*

Básicamente, el método ordenado **.sort()** muta o CAMBIA LA LISTA mientras que la función de ordenación **sorted()** retorna una copia ordenada de la lista y NO CAMBIA LA LISTA la lista original:

.sort() sólo funciona con LISTAS. Ordena alfabéticamente y realiza cambios en la secuencia original, pero no devuelve nada. Si intentamos guardarlo en una variable nos devolvería NONE.

```
num_decimales = [2.1, 1.84, 6.45, 1.03, 3.25, 0.42]
```

a) Devolviendo una lista ordenada de forma ascendente:

```
num_decimales.sort()

print(num_decimales)          #devuelve [0.42, 1.03, 1.84, 2.1, 3.25, 6.45]
```

b) Devolviendo una lista ordenada de forma descendente:

```
num_decimales.sort(reverse = True)
print(num_decimales)          #devuelve [6.45, 3.25, 2.1, 1.84, 1.03, 0.42]
```

En a) y en b), la lista queda MODIFICADA según nuestro último cambio.

.sorted() nos permite ordenar los datos de una determinada manera (ascendente, descendiente), SIN afectar a la lista original:

```
num_decimales = [2.1, 1.84, 6.45, 1.03, 3.25, 0.42]
print(sorted(num_decimales, reverse = True))  #devuelve [6.45, 3.25, 2.1, 1.84, 1.03, 0.42]
```

Nuestra lista original quedaría intacta:

```
print(num_decimales)          #devuelve [2.1, 1.84, 6.45, 1.03, 3.25, 0.42]
```

5. *¿Qué es un operador de reasignación?*

Estos operadores reasignan el valor de la variable mediante una SINTAXIS SIMPLIFICADA. Por ejemplo, tenemos una variable var cuyo valor de 250, y lo queremos incrementar en 50. En ambos casos, el resultado es el mismo, el valor final de var es 300, pero la segunda forma es obviamente más simple:

```
var = 250
```

Forma larga	*Forma simple*
var = var + 50	var += 50

Los otros operadores de reasignación son:

-=	var **-=** 50	resta	#devuelve **200**
***=**	var ***=** 50	multiplicación	#devuelve **12500**
/=	var **/=** 50	división	#devuelve **5.0 (**convierte a **"*float*")**
//=	var **//=** 50	división de enteros	#devuelve **5 (**toma el resultado de número entero o ***floor*****)**
%=	var **%=** 51	módulo	#devuelve **46 (**resto de la división**)**
****=**	var ****=** 3	exponente	#devuelve **15625000 (**250x250x250**)**