Course Checkpoint 4 - Preguntas teóricas

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

Tanto las listas como las tuplas son colecciones ordenadas de elementos. Las listas usan corchetes "[]" y las tuplas paréntesis "()" para almacenar esos elementos. La diferencia más importante entre ellas es que las listas son mutables y las tuplas son inmutables. Es decir, en el caso de las listas, al ser mutables, se puede modificar el contenido una vez se hayan creado, por ejemplo usando funciones de append() para añadir elementos, pop() para eliminarlos, sort() para ordenarlos alfabéticamente, etc. Por lo contrario, en una tupla no se pueden cambiar los contenidos después de haberla creado, sino que habría que realizar una reasignación, pero no se pueden usar funciones como append() o pop() para modificar los elementos. Lo que si que se puede hacer, al igual que en las listas, es acceder a sus elementos, por ejemplo a través de un index() o un slice().

Por ejemplo:

```
# Esto es una lista
lista_compra = ['tomates', 'patatas', 'manzanas']

# Esto es una tupla
tupla_compra = ('tomates', 'patatas', 'manzanas')
```

2. ¿Cuál es el orden de las operaciones?

Para saber el orden de las operaciones está el acrónimo de PEMDAS, donde se muestra la jerarquía determinan cuáles se realizan en primera instancia:

P = Paréntesis. La operación que tiene mayor prioridad es la que se encuentra dentro de paréntesis.

E = Exponente. La segunda más importante sería un exponente, es decir, un "**" o un pow().

M y D = **M**ultiplicación y **D**ivisión. Estas dos operaciones son equivalentes después de un exponente, por lo tanto, si se encuentran las dos a la vez se realizan las operaciones de izquierda a derecha.

A y S = **A**dición y **S**ustracción. Las sumas y las restas serían las operaciones de menor prioridad en igualdad de condiciones como las dos anteriores, por lo que, de la misma forma, en caso de encontrarnos con las dos, la operación se realiza de izquierda a derecha.

Por ejemplo:

```
# Esto es una operación matemática

operacion = 2 * 3 + (5-2) - 2**2 + 4 // 2
```

- Paso 1, paréntesis: (5-2) = 3

```
# Resultado del paso 1

operacion = 2 * 3 + 3 - 2**2 + 4 // 2
```

- Paso 2, exponente: 2**2 = 4

```
# Resultado del paso 2

operacion = 2 * 3 + 3 - 4 + 4 // 2
```

- Paso 3 multiplicación y división. Como aquí tenemos ambos, se hace de izquierda a derecha:
    - 2*3 = 6
    - 4//2 = 2 (esto es una división entera, aunque en este caso el número resultante ya es entero)

```
# Resultado del paso 3.1

operacion = 6 + 3 - 4 + 2
```

- Paso final, sumas y restas, que también se realizan de izquierda a derecha:
    - 6+3 = 9

```
# Resultado del paso 3.2

operacion = 9 - 4 + 2
```

    - 9-4 = 5

```
# Resultado del paso 3.3

operacion = 5 + 2
```

RESULTADO DE LA OPERACIÓN = 7

```
operacion = 2 * 3 + (5-2) - 2**2 + 4 // 2

print(operacion)
✓ 0.0s
7
```

3. ¿Qué es un diccionario Python?

Un diccionario es un almacén de elementos que guarda claves junto a valores dentro de las llaves "{}". Estas claves en un mismo diccionario son únicas y dentro se pueden almacenar uno o más valores asociados. Las claves y sus valores están separados con dos puntos y los distintos conjuntos de clave-valor con comas.

Al igual que las listas, los diccionarios son mutables ya que se pueden añadir o quitar estas clave-valor después de haber creado el diccionario.

Por ejemplo:

```
# Esto es un diccionario

persona_1 = {
   "nombre": "Andrea",
   "edad": 27,
   "estudios": [
      "Quimica",
      "Python"
   ]
}
```

Como he mencionado, es mutable, por lo que se puede por ejemplo añadir nuevos conjuntos clave:valor, nuevos valores en una clave ya creada etc (o eliminar).

Por ejemplo, vamos a añadir "Full Stack" en la clave estudios usando el método de append() para añadirlo al final de la lista:

```
# Primero entra dentro de la clave estudios, después añade al final de la lista "Full Stack"

persona_1["estudios"].append("Full Stack")
```

El resultado al imprimir sería el siguiente. Como se observa, se ha adicionado después de "Python" el nuevo valor de "Full Stack":

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

Estos métodos son sort() (método ordenado) y sorted() (función de ordenación). Ambos se utilizan para ordenar, aunque sort() es para listas y sorted() se puede usar en distintos tipos de datos como tuplas o diccionarios por ejemplo. En los dos casos se puede ordenar

alfabéticamente si son strings o en orden ascendente si son números, o en contra usando reverse = True, es decir, de Z-A en strings y de forma descendente en números.

Las diferencias son las siguientes:

- Como he mencionado anteriormente, sort() aplica a listas y sorted() a más tipos de datos como tuplas, diccionarios, listas etc.
- El método sort() modifica la lista ordenada tal cual, es decir, cambia la lista ya creada. En cambio, la función de sorted() no modifica la lista (u otro tipo de dato) directamente, sino que crea una nueva lista y la devuelve de forma ordenada.
- El método sort() no devuelve un valor, es decir, si yo intento hacer un print de una lista.sort() me va a devolver None, ya que este método modifica la lista in situ. En cambio, esto sí se puede realizar usando sorted(). Esto lo podemos ver visualmente en el siguiente ejemplo:

```
shopping_list = ['potatoes', 'tomatos', 'milk']
```
✓ 0.0s

```
# Aquí cambia el orden de la lista original

shopping_list.sort()
print(shopping_list)
```
✓ 0.0s

```
['milk', 'potatoes', 'tomatos']
```

```
# Si intento imprimirla ordenada sin cambiarla va a devolver None
print(shopping_list.sort())
```
✓ 0.0s

```
None
```

```
# Aquí cambia el orden pero sin modificar la lista original

print(sorted(shopping_list))
```
✓ 0.0s

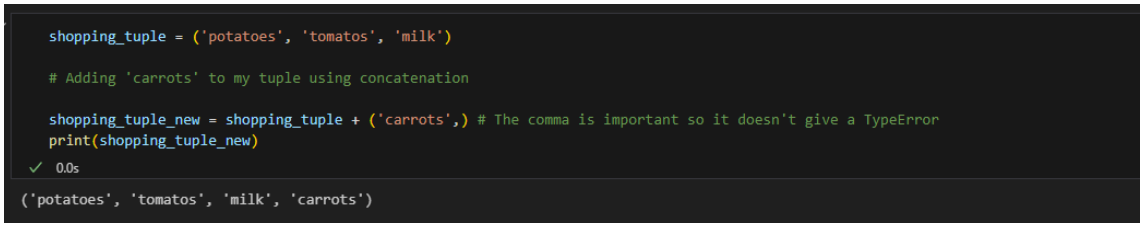
```
['milk', 'potatoes', 'tomatos']
```

```
# Pero no cambia la lista original al realizar la operación como con sort()

sorted(shopping_list) # Habría que guardarlo en una nueva variable nueva

print(shopping_list)
```
✓ 0.0s

```
['potatoes', 'tomatos', 'milk']
```

5. ¿Qué es un operador de reasignación?

Un operador de reasignación se usa para poder añadir un elemento a una tupla ya creada. Como anteriormente he mencionado, una tupla es inmutable, por lo que no se puede añadir por ejemplo usando append(), en este caso se debe hacer una reasignación que se realiza con una concatenación. Como, por ejemplo, en el ejercicio 9 de M2C4 donde he añadido el elemento de 'carrors' al final de la tupla mediante concatenación:



```
shopping_tuple = ('potatoes', 'tomatos', 'milk')

# Adding 'carrots' to my tuple using concatenation

shopping_tuple_new = shopping_tuple + ('carrots',) # The comma is important so it doesn't give a TypeError
print(shopping_tuple_new)
```

✓ 0.0s

```
('potatoes', 'tomatos', 'milk', 'carrots')
```