CHECKPOINT 4 – EJERCICIO TEÓRICO

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

La primera diferencia es la sintaxis de las mismas, es decir como se redactan. Mientras que las listas están delimitadas por corchetes [ ], las tuplas lo están por paréntesis ( ). Ejemplo:

```
tuple_example = (1, 2, 3)
list_example = [1, 2, 3]
```

La diferencia más importante es que las listas son mutables, es decir, significa que, si hacemos un cambio a nuestra lista, ese cambio es permanente así que no tenemos que reasignarlo a un valor. En cambio, las tuplas son inmutables, es decir, no se pueden hacer cambios por lo que, las diferentes técnicas que usaremos para eliminar elementos en realidad no eliminarán elementos de la tupla original. Hay que tener muy en cuenta esto de cara a elegir con que trabajar si con tuplas o listas, si se pretende realizar cambios tendrán que ser listas y si en cambio sabemos con seguridad que no deseamos realizar cambios lo mejor es trabajar con tuplas.

No obstante, existen formas alternativas de llevar a cabo cambios en las tuplas. Podemos añadir elementos a una tupla reasignando la variable, de forma que no cambiamos la tupla en origen, sino que asignamos un nuevo valor a la variable existente. Por ejemplo:

```
tuple_change += (4,)
```

Para que Phyton detecte que es una tupla es importante poner una coma después del dato añadido, dentro del paréntesis. Si no se pone la coma lo detecta como una simple string y daría error porque no se puede meter una string en una tupla.

2. **¿Cuál es el orden de las operaciones?**

El orden de operaciones en Python es el mismo que para cualquier otra expresión matemática. El orden sería el siguiente, de mayor a menor prioridad:

I. Paréntesis
II. Exponentes
III. Multiplicaciones
IV. Divisiones
V. Sumas
VI. Restas

Esto significa que el programa primero ejecutará el código que se encuentre entre paréntesis, luego los exponentes y así sucesivamente. En las guías se establece la regla nemotécnica “Please excuse my dear aunt Sally” para recordar PEMDAS, que corresponde a las iniciales de las operaciones en inglés.

Especificar también que hay mucha controversia entre que es primero si es multiplicación o división, pero se han hecho pruebas y el resultado no varía por lo que no es algo importante.

3. ¿Qué es un diccionario Python?

Un diccionario es un almacén de datos con valores clave, eso significa que se puede almacenar en una variable y se puede crear una clave con un valor correspondiente. Tiene un conjunto de claves y luego tiene valores asociados para cada una de esas claves, es decir, en lugar de incluir un conjunto de valores individuales, incluye un conjunto de key value pairs (pares clave valor), que consisten en una determinada clave y un valor asignado a esta clave. Por ejemplo:

```
diccionary_example = {
   'uno’: 1,
   'dos': 2,
}
```

Solo pueden tener una relación de tipo valor de clave único uno a uno, es frecuente que las claves del diccionario se traten de strings, pero pueden contener cualquier otro tipo de estructura de datos, pudiera contener una lista, una tupla, otro diccionario, una cadena, un número, etc.

Del mismo modo, los diccionarios pueden estar anidados dentro de otras colecciones como listas, tuplas u otros diccionarios. Por ejemplo:

```
nested_diccionary = {
  "numeros": [ 1, 2, 3 ],
  "letras": ["a", "b", "c"],
}
```

Podemos hacer queries con una sintaxis similar a la usada en las listas, pero en lugar del índice debemos usar la clave para obtener su correspondiente valor. Por ejemplo:

```
print(nested_diccionary["letras"])
```

Un elemento importante a tener en cuenta es que, si no se encuentra el valor buscado, obtendremos un error. Esto puede ser lo deseable en algunos casos, pero tenemos también la opción de utilizar la función get()

para establecer un valor por defecto que se nos devuelva si la clave introducida no se encuentra en el diccionario. Get() nos permite tener múltiples procesos y las verificaciones múltiples suceden automáticamente. Por ejemplo:

```
diccionary_nested.get(“e", 'letter not found')
```

Se considera una práctica óptima asegurarnos de cubrir todas las posibilidades que tengan una situación así, donde buscamos una clave que tal vez exista o no y que haya alguna respuesta automática. Esto les brindará reacciones instantáneas para avisarles que lo que intentaron buscar no existen dentro del diccionario mencionado.

Además de esto, los diccionarios son mutables, lo cual significa que podemos añadir, editar o eliminar key value pairs. Por ejemplo:

```
diccionary_example['tres'] = 3 → Ejemplo de adición
diccionary_example['uno'] = 0 → Ejemplo de edición
del diccionary_example ['uno'] → Ejemplo de eliminación
```

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

Entiendo que cuando nos referimos al método ordenado y a la función de ordenación, estamos refiriéndonos a sort() y sorted().

sort() ordena todos los elementos de forma alfabética. En el caso de palabras (strings) las ordena de la A-Z, en caso de fechas de lo más antiguo a lo más reciente y en caso de números de menor a mayor. Por ejemplo:

```
languages = ['english', 'arabic', 'spanish', 'russian']
languages.sort()
```

También hay una manera de utilizar sort () para ordenar todos los elementos en orden opuesto de la Z-A y de más reciente a más antiguo y en números de mayor a menor. Por ejemplo:

```
languages.sort(reverse=True)
```

sorted() en cambio, almacena en una variable la lista ordenada. Tiene el mismo comportamiento que sort() excepto que permite almacenar ese valor en otra variable diferente dejando la lista original totalmente intacta, sin cambios. Por ejemplo:

```
sorted_languages = sorted(languages)
```

Como en el caso de sort() también se puede ordenar de mayor a menor. Por ejemplo:

```
sorted_languages = sorted(languages , reverse=True)
```

Por lo tanto, podemos determinar que la principal diferencia entre ambas funciones es que sort() cambia la lista original cambiando la estructura de todos los elementos. Además sort() o almacena nada, solo ordena la lista de una determinada manera, cambia las etiquetas pero no devuelve un valor, por lo que si se intenta crear una nueva variable para almacenar los datos que devuelva, devolvería none.

En cambio sorted() no cambiaría la lista original, pero sí retornaría una nueva lista ordenada, de forma que podamos trabajar tanto con la lista intacta como con la nueva ordenada de una determinada manera.

5. ¿Qué es un operador de asignación?

Los operadores de asignación son una herramienta importante en Python para modificar el valor de las variables. En lugar de tener que redefinir la variable cada vez que se modifica su valor, los operadores de asignación permiten modificarla directamente, por lo que nos permiten realizar una operación y almacenar su resultado en la variable inicial.

El operador básico de asignación en Python es el signo igual (=), que se utiliza para asignar un valor a una variable. Sin embargo, Python también tiene algunos operadores de asignación que nos permiten realizar operaciones matemáticas y asignar el resultado a una variable en un solo paso. Estos operadores son += , -= , /= , //= , %= , *=, **=

Habitualmente se usa para operaciones de suma y resta, o incremento/decremento de variables. Por ejemplo, imaginemos que queremos incrementar una variable según recibimos datos:

```
total += 10
```

Esta última operación sería equivalente a:

```
total = total + 10
```