CheckPoint 4 – Preguntas teóricas

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

La lista y la tupla son estructuras de tipos de datos cuya principal diferencia es que una lista es mutable, es decir, se puede modificar, y una tupla es inmutable, no se pueden modificar los valores de sus elementos.

Es interesante usar listas cuando vamos a necesitar modificar los datos, es decir, añadir, eliminar, cambiar de posición de los elementos… Python permite realizar operaciones sobre las listas, que sobre las tuplas no están permitidas. Por ejemplo, para las listas podemos añadir elementos usando las funciones inset(), append(), extend(). Se pueden eliminar elementos usando las funciones remove(), pop() y del. Se pueden ordenar usando la función sort() y sorted().

Sin embargo, conviene usar tuplas cuando vamos a necesitar que los datos de la colección no sean modificados.

La sintaxis de una lista es:

```
mi_lista = [‘elemento1’, ‘elemento2’, ‘elemento3’]
```

La sintaxis de una tupla es:

```
mi_tupla = (‘elemento1’, ‘elemento2’, ‘elemento3’)
```

¿Cuál es el orden de las operaciones?

En matemáticas el orden de las operaciones afecta al resultado de los cálculos. Existe un orden para realizarlas que puede recordarse usando el acrónimo PEMDAS. La jerarquía es la siguiente: Paréntesis, Exponentes, Multiplicación, División, Adición y Sustracción.

¿Qué es un diccionario Python?

Un diccionario es un tipo de estructura de datos mutable utilizada en Python. Consiste en una colección de elementos que almacena datos con pares clave-valor en una variable. La sintaxis para crear un diccionario en Python es la siguiente:

```
mi_diccionario = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3",
}
```

Para acceder a los valores del diccionario usamos las claves. Por ejemplo:

```
diccionario = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3",
}

par2 = diccionario['key2']
print(par2)
value2
```

Son estructuras dinámicas, es decir, se pueden añadir o eliminar elementos del diccionario.

Los diccionarios pueden ser anidados, es decir, que pueden contener a otro diccionario en su campo valor. O pueden contener listas anidadas cuyos valores pueden consultarse. Por ejemplo:

```
diccionario = {
  "key1": ["value11", "value12", "value13"],
  "key2": "value2",
  "key3": "value3"
}

print(diccionario['key1'])
['value11', 'value12', 'value13']

print(diccionario['key1'][1])
value12
```

También podemos añadir elementos al diccionario de la misma manera en que se realiza una consulta.

```
diccionario = {
  "key1": ["value11", "value12", "value13"],
  "key2": "value2",
  "key3": "value3"
}
diccionario["key4"] = ["value41", "value42"]
print(diccionario)
{
'key1': ['value11', 'value12', 'value13'],
'key2': 'value2',
'key3': 'value3',
'key4': ['value41', 'value42']
}
```

La función **.get()** sirve para establecer un valor por defecto con dos argumentos: la clave que buscamos que existir o no y la respuesta automática. Ejemplo:

```
mi_diccionario = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
busqueda1 = mi_diccionario.get("key1", "No key")
print(busqueda1)
value1
busqueda2 = mi_diccionario.get("key4", "No key")
print(busqueda2)
No key
```

Los 'Dictionary View Objects' permiten seleccionar y ver dinámicamente los valores, las claves y todos los diferentes elementos dentro de los diccionarios mediante las funciones .keys(), .values() y .items(). Se obtienen los objetos dict.keys(), dict.values() y dict.items() con la consulta solicitada.

```
mi_diccionario = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
print(mi_diccionario.keys())
dict_keys(['key1', 'key2', 'key3'])

print(mi_diccionario.values())
dict_values(['value1', 'value2', 'value3'])

print(mi_diccionario.items())
dict_items([('key1', 'value1'), ('key2', 'value2'), ('key3', 'value3')])
```

La función **.len()** permite ver el numero de elementos de un diccionario.

```
mi_diccionario = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}

mi_diccionario_items = mi_diccionario.items()
print(len(mi_diccionario_items))
3
```

La función **.copy()** da la capacidad de realizar cualquier acción que desees y teniendo la certeza de que los datos con los que estás trabajando no han cambiado.

```
mi_diccionario = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}

mi_diccionario = list(mi_diccionario.copy().values())
print(mi_diccionario)
['value1', 'value2', 'value3']
```

Con **list()** podemos convertir un diccionario en una lista y acceder a valores del diccionario. Ejemplo:

```
mi_diccionario = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}
mi_diccionario_items = mi_diccionario.items()
print(list(mi_diccionario_items)[1])
('key2', 'value2')
```

Con las funciones **del** y **.pop()** podemos eliminar elementos de un diccionario. Si el elemento no existe .pop() devuelve un valor no un error.

```
mi_diccionario = {
  "key1": "value1",
  "key2": "value2",
  "key3": "value3"
}

del mi_diccionario["key1"]
print(mi_diccionario)
{'key2': 'value2', 'key3': 'value3'}
del mi_diccionario["key4"]
print(mi_diccionario)
KeyError: 'key4'
```

```
mi_diccionario.pop("key3", "Not found")
print(mi_diccionario)
{'key1': 'value1', 'key2': 'value2'}
```

```
removed_item = mi_diccionario.pop("key4", "Not found")
print(removed_item)
Not found
```

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

Las listas se pueden **ordenar** por orden alfabético con la función **.sort()** en Python. Si además de ordenar se quiere invertir el orden se debe pasar el argumento **reverse=True**. La clasificación con .sort() ordena y/o invierte los elementos de la lista pero no devuelve ningún valor, no los almacena como una operación estándar dentro de una variable.

```
mi_lista = [ 'elemento3', 'elemento1', 'elemento2']
mi_lista.sort()
print(mi_lista)
['elemento1', 'elemento2', 'elemento3']
mi_lista.sort(reverse=True)
print(mi_lista)
['elemento3', 'elemento2', 'elemento1']
```

La función **sorted()** en Python sirve para ordenar listas en Python y permite almacenar la lista ordenada en una variable diferente. La lista desordenada, la original, permanece inalterada.

```
mi_lista = ['elemento1', 'elemento2', 'elemento5', 'elemento4', 'elemento3']
mi_lista_ordenada = sorted(mi_lista)
print(mi_lista)
['elemento1', 'elemento2', 'elemento5', 'elemento4', 'elemento3']
print(mi_lista_ordenada)
['elemento1', 'elemento2', 'elemento3', 'elemento4', 'elemento5']
```

¿Qué es un operador de reasignación?

Un operador de reasignación permite realizar una operación cualquiera y almacenar el resultado en una variable. Los operadores de reasignación son los siguientes:

= Asigna a la variable de la izquierda el contenido que aparece a la derecha de =

x = 1

=+ Sumar y asignar el resultado a la variable inicial.

x +=1 equivale a x = x + 1

-= Restar y asignar el resultado a la variable inicial.

x -= 1 equivale a x = x - 1

*= Multiplicar una variable por otra y almacenar el resultado en la primera.

x *=1 equivale a x = x * 1

/= Dividir una variable por otra y almacenar el resultado en la primera.

x /=1 equivale a x = x / 1

%= Módulo de la división de dos variables y almacenar su resultado en la primera.

x %=1 equivale a x = x % 1

//= Cociente entre dos variables y almacena el resultado en la primera.

x //=1 equivale a x = x // 1

**= Exponente del primer número elevado al segundo, y almacena el resultado en la primera variable.

x **=1 equivale a x = x ** 1