CHECKPOINT 4

QUESTION 1

Statement

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

Answer

La principal diferencia entre una lista y una tupla es la mutabilidad:

- Las listas son mutables, lo que significa que sus elementos pueden cambiar después de la creación.
- Las tuplas son inmutables, lo que significa que sus elementos no pueden modificarse una vez creadas.

Las listas son útiles cuando se necesita modificar los datos dinámicamente, agregar o eliminar elementos. Por ejemplo, cuando tenemos una série de elementos que entran y salen de la colección continuamente.

Las tuplas son más eficientes en cuanto a memoria y se usan cuando los datos no deben cambiar. Esto puede darse cuando estamos trabajando con datos que no generamos nosotros, como las siglas de los países para una aplicación de busqueda de vuelos en aerolineas.

Experiencia personal: En mi caso, he trabajado con C#, y existe una equivalencia que sería Array:Tupla ; List:Lista. En videojuegos las listas son colecciones clave, como por ejemplo para manejar la entrada y salida de enemigos en el campo de visión del jugador. Tienen un sinfin de usos en comparación con Arrays tradicionales.

Examples

```
# Modificación de lista y tupla
lista = [1, 2, 3]
lista.append(4, 5, 7)
lista[5] = 6

tupla = (1, 2, 4)
tupla[2] = 3  # Error
```

```
# Sistema de detección en videojuego con listas dinámicas
import schedule
import time
import random
import math

# Posición del jugador (x, y)
player_position = (0, 0)

# Lista de enemigos dispersados en el mapa
enemies = [{"id": i, "position": (random.uniform(-50, 50), random.uniform(-50, 50))} for i in

def calculate_distance(p1, p2):
    """Calcula la distancia euclidiana entre dos puntos (x, y)."""
    return math.sqrt((p2[0] - p1[0]) ** 2 + (p2[1] - p1[1]) ** 2)

def check_enemies():
    """Detecta enemigos dentro de un rango de 20 metros."""
    detected_enemies = []
```

```
    for enemy in enemies:
        distance = calculate_distance(player_position, enemy["position"])
        if distance < 20:
            detected_enemies.append(enemy)

def update():
    check_enemies()
    if detected_enemies:
        print(f"Enemigos detectados ({len(detected_enemies)}): {[e['id'] for e in detected_ene

schedule.every(0.1).seconds.do(update)

# Bucle principal de detección
while True:
    schedule.run_pending()
    time.sleep(0.1)
```

QUESTION 2

Statement

¿Cuál es el orden de las operaciones?

Answer

El orden de las operaciones en Python sigue la jerarquía PEMDAS: Paréntesis, Exponentes, Multiplicación/División, Adición/Sustracción. Esta regla se utiliza en operaciones matemáticas, programación, ingeniería, economía...

- Paréntesis: se evalúan primero.
- Exponentes: se evalúan a continuación.
- Multiplicación/División: se evalúan justo después
- Adición/Sustracción: se evalúan al final.

Examples

```
 # Ejemplo de operaciones matemáticas escalonadas
x = 3 + 2 * (8 / 4) ** 2
x = 3 + 2 * (2) ** 2
x = 3 + 2 * 4
x = 3 + 8
x = 11
```

QUESTION 3

Statement

¿Qué es un diccionario Python?

Answer

Un diccionario en Python es una colección de elementos, donde cada uno tiene una llave key y un valor value, lo que se conoce como estructura clave-valor.

Una caracteristica fundamental de los diccionarios es que el value puede ser alfanumérico, e incluso nulo, por lo que es muy versatil.

Por ejemplo, un JSON usa esta estructura clave-valor. Este tipo de estructuras se usan también en bases de datos no relacionales, como MongoDB. Entre las principales ventajas de los diccionarios podemos encontrar que:

- Son dinámicos, pueden crecer o decrecer, se pueden añadir o eliminar elementos.
- Son indexados, los elementos del diccionario son accesibles a través del key.
- Son anidados, un diccionario puede contener otras colecciones como listas o diccionarios.

Examples

```python
# Diccionario con los stats de diferentes enemigos
enemies = {
    "Goblin": {
        "HP": 30,
        "Attack": 5,
        "Defense": 2,
    },
    "Dragon": {
        "HP": 300,
        "Attack": 50,
        "Defense": 20,
    }
}

# Modificación dinámica
enemies["Skeleton"] = {
    "HP": 10,
    "Attack": 10,
    "Defense": 4,
}
enemies["Dragon"]["Magic"] = ["Fireball", "Ice Storm", "Thunderbolt"]

# Acceso a datos
print(enemies["Goblin"])  # {'HP': 30, 'Attack': 5, 'Defense': 2}
print(enemies["Skeleton"]["HP"])  # 10
print(enemies["Dragon"]["Magic"][0])  # "Fireball"
```

QUESTION 4

Statement

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

Answer

La diferencia principal entre el método `sort()` y la función `sorted()` radica en cómo afectan la lista original, y la decisión de usar uno u otro depende de si queremos conservar la lista original o no:

- `sort()`: Es un método de las listas en Python. Modifica la lista original en el lugar (es decir, ordena la lista directamente), y no devuelve un valor. Esto significa que después de llamar a `sort()`, la lista original estará ordenada, y el cambio es irreversible.
- `sorted()`: Es una función incorporada en Python que devuelve una nueva lista ordenada, dejando la lista original sin modificar. Esto es útil cuando necesitamos mantener la lista original intacta mientras trabajamos con una versión ordenada de la misma.

Examples

```
# Sort()

myList = [3, 1, 4, 1, 5, 9, 2, 6]
myList.sort()  # Lista original ordenada
print(myList)  # [1, 1, 2, 3, 4, 5, 6, 9]
```

```
# Sorted()

myList = [3, 1, 4, 1, 5, 9, 2, 6]
myNewList = sorted(myList)  # Nueva lista ordenada
print(myNewList)  # [1, 1, 2, 3, 4, 5, 6, 9]
print(myList)  # [3, 1, 4, 1, 5, 9, 2, 6]
```

QUESTION 5

Statement

¿Qué es un operador de reasignación?

Answer

Un operador de reasignación es un tipo especial de operador que permite modificar el valor de una variable utilizando su valor actual. Estos operadores son útiles para simplificar el código y realizar operaciones en una sola línea sin tener que escribir la variable dos veces. Los operadores de reasignación más comunes son:

- `+=` : Suma el valor de la derecha a la variable. Ejemplo: `x += 5` es igual a `x = x + 5`.
- `-=` : Resta el valor de la derecha a la variable. Ejemplo: `x -= 3` es igual a `x = x - 3`.
- `*=` : Multiplica la variable por el valor de la derecha. Ejemplo: `x *= 2` es igual a `x = x * 2`.
- `/=` : Divide la variable entre el valor de la derecha. Ejemplo: `x /= 4` es igual a `x = x / 4`.
- `%=` : Calcula el residuo de la división de la variable entre el valor de la derecha. Ejemplo: `x %= 3` es igual a `x = x % 3`.
- `**=` : Eleva la variable a la potencia del valor de la derecha. Ejemplo: `x **= 2` es igual a `x = x ** 2`.
- `//=` : Realiza una división entera entre el valor de la derecha. Ejemplo: `x //= 2` es igual a `x = x // 2`.

Estos operadores son muy utilizados en programación para optimizar el código y evitar la necesidad de escribir expresiones más largas. Además, son especialmente útiles en bucles o en la manipulación de datos donde el valor de una variable se va modificando repetidamente durante la ejecución del programa.

Examples

```python
 # Equivalencias de operadores de reasignación
x = 10
x += 5 # x = x + 5
x -= 3 # x = x - 3
x *= 2 # x = x * 2
x /= 4 # x = x / 4
x %= 3 # x = x % 3
x **= 2 # x = x ** 2
x //= 2 # x = x // 2
print(x)  # 12
```

```python
# Ejemplo de cálculo de posición en un videojuego
import time

# Posición, velocidad y aceleración en 2D (x, y)
position = [0, 0]
velocity = [3, 2]
acceleration = [0.5, 0.2]

# Tiempo de actualización
delta_time = 0.1

# Simulación durante 3 segundos
for frame in range(30):
    # Actualizar velocidad con la aceleración
    velocity[0] += acceleration[0] * delta_time
    velocity[1] += acceleration[1] * delta_time
```

```
    # Actualizar la posición con la velocidad
    position[0] += velocity[0] * delta_time
    position[1] += velocity[1] * delta_time

    print(f"Frame {frame+1}: Posición = {position}, Velocidad = {velocity}")
```