Guía de conceptos en ROS 2 y Python

1. ¿Qué es un Nodo en ROS 2?

Imagina un sistema de robótica como una ciudad. En esa ciudad, cada edificio o servicio importante (un hospital, una estación de policía, un semáforo) es un programa independiente que hace una tarea específica.

En ROS 2, cada uno de esos programas independientes que se comunican entre sí se llama **Nodo**.

- Tu NodoCamaraTello es un nodo: se encarga de hablar con el Tello y publicar sus imágenes y recibir comandos.
- Tu NodoProcesadorImagen es otro nodo: se encarga de coger esas imágenes, procesarlas (detectar colores, calcular distancias) y publicar los resultados.
- Tu NodoControlLogica es un tercer nodo: recibe los resultados de la imagen y toma decisiones sobre cómo debe moverse el Tello.

Cada nodo tiene un nombre único (como 'nodo_camara_tello') para que ROS 2 sepa quién es quién.

2. Entendiendo las Clases en Python (para programadores Java)

Si vienes de Java, el concepto de **Clase** te es muy familiar: es un **plano o plantilla** para crear objetos. Define las características (variables o "atributos") y los comportamientos (funciones o "métodos") que tendrán los objetos que crees a partir de ella.

Similitudes con Java:

- Ambos lenguajes usan clases para agrupar datos y funciones.
- Para usar una clase, necesitas crear una **instancia** (un objeto) de esa clase.

Diferencias Clave con Java:

Característica	Java	Python
Creación de Objeto	Usa new: MiClase miObjeto = new MiClase();	No usa new: mi_objeto = MiClase()
Modificadores	public, private, protected explícitos	Todo es público por defecto (convención: _ o __)

Tipado	Tipado estático (String nombre;)	Tipado dinámico (no declaras el tipo de la variable)

3. El Misterio de self

Esta es la parte que más suele "chocar" si vienes de Java, pero es sencilla cuando la entiendes.

En Python, self es una convención (un nombre que se usa por acuerdo en la comunidad, aunque técnicamente podrías llamarlo de otra forma, ¡no lo hagas!) que se refiere a la **instancia actual del objeto** sobre el cual se está llamando un método.

Piensa en self como el this de Java, pero con una peculiaridad:

- En Java, this es implícito: lo usas dentro de un método de instancia para referirte al objeto actual, pero no lo declaras como un parámetro.
- En Python, self es **explícito**: **siempre debe ser el primer parámetro** de cualquier método que pertenezca a la clase (un método de instancia).

Ejemplo Sencillo:

```
class Robot:
    def __init__(self, nombre):
        # 'self.nombre' es un atributo de ESTA instancia del Robot
        self.nombre = nombre
        self.bateria = 100

    def mostrar_estado(self):
        # Para acceder a 'nombre' o 'bateria' de ESTA instancia, usamos 'self.'
        print(f"Robot {self.nombre} tiene {self.bateria}% de batería.")

    def recargar(self, energia):
        # 'self' permite modificar 'bateria' de ESTA instancia
        self.bateria += energia
        if self.bateria > 100:
            self.bateria = 100
        print(f"{self.nombre} recargado. Nueva batería: {self.bateria}%")

# Creamos una instancia de Robot
r2d2 = Robot("R2D2") # Aquí se llama a __init__(self, "R2D2")
```

```
r2d2.mostrar_estado() # Cuando llamas esto, Python pasa 'r2d2' como 'self'
                      # Internamente es como: Robot.mostrar_estado(r2d2)

r2d2.recargar(20)     # Cuando llamas esto, Python pasa 'r2d2' como 'self' y '20' como 'energia'
                      # Internamente es como: Robot.recargar(r2d2, 20)
r2d2.mostrar_estado()
```

Como ves en el ejemplo de recargar(self, energia), tú solo pasas 20 al llamar r2d2.recargar(20). Python se encarga automáticamente de pasar el objeto r2d2 como el argumento self. Esta es una de las grandes diferencias visuales con Java.

4. El Constructor __init__

El método __init__ (con dos guiones bajos antes y después) es el **constructor** en Python.

- **¿Cuándo se ejecuta?** Automáticamente, cada vez que creas un nuevo objeto (instancia) de la clase.
- **¿Para qué sirve?** Su función principal es preparar el nuevo objeto. Esto significa **inicializar sus atributos** (variables internas) para que el objeto empiece con un estado conocido.

Ejemplo en tu NodoCamaraTello:

```
class NodoCamaraTello(Node):
   def __init__(self): # 'self' es la instancia que se está creando
      super().__init__('nodo_camara_tello') # Llama al constructor de la clase padre (Node)

      self.bridge = CvBridge() # Crea un CvBridge para ESTA instancia
      self.tello = Tello(host=TELLO_IP) # Crea un objeto Tello para ESTA instancia
      self.frame_reader = None # Inicializa esta variable para ESTA instancia
      # ... y muchas otras inicializaciones para 'self'
```

Dentro de __init__, usas self. para crear los atributos que pertenecerán específicamente a ese NodoCamaraTello recién creado.

5. rclpy: El Sistema Nervioso de ROS 2

rclpy es la biblioteca de Python que te permite que tu código hable y funcione dentro del ecosistema de ROS 2. Es como el **"sistema operativo" o "sistema nervioso"** de tu nodo.

- rclpy.init(args=args): Es el comando para **iniciar** ese sistema nervioso de ROS 2 en tu programa. Siempre va al principio.
- rclpy.spin(tu_nodo): Es el **corazón** de tu nodo. Le dice al sistema nervioso de ROS 2: "Mantente activo, escucha todo lo que tengas que escuchar y ejecuta las funciones (callbacks) cuando ocurran los eventos". Es un bucle infinito que se encarga de todo sin que tú tengas que escribir un while True.
- rclpy.shutdown(): Es el comando para **apagar** el sistema nervioso de ROS 2 de forma limpia al final de tu programa.

Cómo rclpy gestiona la comunicación:

rclpy se encarga de que tus nodos envíen y reciban información de forma ordenada:

a) Suscriptores: Recibiendo Información (Oídos del Nodo)

- **Propósito:** Un suscriptor es como los **oídos** de tu nodo. Le permite escuchar mensajes que otros nodos están publicando en un topic específico.
- **Creación (create_subscription):**
  self.suscriptor_comandos_velocidad = self.create_subscription(
  Float32MultiArray, # Tipo de mensaje que esperamos
  ROS_TOPIC_COMANDOS_VELOCIDAD, # El "canal" que vamos a escuchar
  self.callback_comandos_velocidad, # La función que se ejecutará cuando llegue un mensaje
  qos_profile_cmd # Reglas de calidad (ej. si es importante que lleguen todos los mensajes)
)

  Aquí le dices a rclpy: "Quiero que **este nodo** (self) escuche en el canal /tello/comandos_velocidad y, cada vez que llegue un mensaje de tipo Float32MultiArray, llama a mi función callback_comandos_velocidad".
- **El callback:** self.callback_comandos_velocidad es una función que se ejecuta **automáticamente** cada vez que llega un nuevo mensaje al topic al que está suscrito. rclpy le pasa el mensaje recibido como argumento.
  - **Analogía:** Imagina que rclpy es un cartero. El cartero (rclpy) está esperando en la oficina. Cuando llega una carta (msg), el cartero corre a tu casa y te la entrega directamente en el método callback_comandos_velocidad. Tú no tienes que ir a buscarla; el cartero la trae.

b) Publicadores: Enviando Información (Boca del Nodo)

- **Propósito:** Un publicador es como la **boca** de tu nodo. Le permite enviar mensajes a un topic para que otros nodos que estén suscritos a ese topic puedan recibirlos.
- **Creación (create_publisher):**
  ```
  self.publicador_imagen_raw = self.create_publisher(
      Image, # Tipo de mensaje que vamos a enviar
      ROS_TOPIC_IMAGEN_RAW, # El "canal" por donde vamos a hablar
      qos_profile_img # Reglas de calidad
  )
  ```

  Aquí le dices a rclpy: "Quiero que **este nodo** (self) pueda hablar por el canal /tello/imagen_raw enviando mensajes de tipo Image".
- **Publicación (publish):**
  ```
  self.publicador_imagen_raw.publish(ros_image_msg)
  ```

  Cuando llamas a publish(), le estás entregando el mensaje a rclpy.
  - **Analogía:** Tú le das la carta (ros_image_msg) al cartero (rclpy), y él se encarga de llevarla a todos los que están suscritos a ese buzón (el topic).

c) Temporizadores: Haciendo Tareas Periódicas (Reloj del Nodo)

- **Propósito:** Un temporizador permite que tu nodo ejecute una función de forma repetida cada cierto intervalo de tiempo. Es como tener un **reloj despertador** interno.
- **Creación (create_timer):**
  ```
  self.timer_camara = self.create_timer(
      TIMER_PERIODO_CAMARA, # El intervalo de tiempo (ej. 1/30.0 segundos)
      self.timer_callback_camara # La función que se ejecutará cada vez
  )
  ```

  Aquí le dices a rclpy: "Quiero que **este nodo** (self) llame a mi función timer_callback_camara cada TIMER_PERIODO_CAMARA segundos".
- **El timer_callback:** self.timer_callback_camara es la función que se ejecuta automáticamente cuando el temporizador se dispara.
  - **Analogía:** rclpy tiene un reloj de cocina. Cuando suena la alarma (cada 1/30 segundos), rclpy te da un "toque" y tú ejecutas la acción definida en timer_callback_camara (que es obtener y publicar un frame de la cámara).

d) Los "Hilos" (Threads) en ROS 2 y Python

Aquí es donde puede haber una pequeña confusión si piensas en los hilos de Java:

- **rclpy.spin() y el Executor:** Por defecto, rclpy.spin() ejecuta todas tus callbacks (de suscriptores, publicadores, temporizadores) de forma **secuencial en un solo hilo**. Esto se llama SingleThreadedExecutor. Es como si tuvieras un solo cerebro, y aunque haga muchas cosas, las hace una detrás de otra muy rápido. Si una callback tarda mucho, puede retrasar las demás.
- **Hilos de djitellopy:** La biblioteca djitellopy que usas para el Tello sí que usa hilos internos para manejar la comunicación con el dron y capturar el vídeo. Esto es importante porque permite que el vídeo se capture continuamente en segundo plano sin bloquear tu nodo de ROS 2. Cuando tú pides self.frame_reader.frame, djitellopy ya lo tiene preparado gracias a su propio hilo interno.

6. Conceptos Adicionales de tu Proyecto

Aquí te explico algunos elementos específicos que aparecen en tus archivos nodoImagen.py y nodoLogica.py:

- **Topics ROS (los "canales de comunicación"):**
  - ROS_TOPIC_IMAGEN_RAW_INPUT = '/tello/imagen_raw': El canal por donde NodoCamaraTello publica las imágenes crudas, y NodoProcesadorImagen las recibe.
  - ROS_TOPIC_IMAGEN_PROCESADA_OUTPUT = '/tello/imagen_procesada': El canal por donde NodoProcesadorImagen publica la imagen con las detecciones dibujadas. Puedes visualizarla con herramientas de ROS 2.
  - ROS_TOPIC_DATOS_DETECCION_OUTPUT = '/tello/datos_deteccion': El canal crucial por donde NodoProcesadorImagen envía los resultados numéricos de la detección (posición del objeto, distancia) a NodoControlLogica.
  - ROS_TOPIC_COMANDOS_VELOCIDAD_OUTPUT = '/tello/comandos_velocidad': El canal por donde NodoControlLogica envía las instrucciones de movimiento al NodoCamaraTello para que este se las transmita al Tello.
- **Mensajes ROS (el "contenido" de la comunicación):**
  - sensor_msgs.msg.Image: Es el tipo de mensaje estándar de ROS 2 para transportar imágenes. Contiene los píxeles de la imagen, su codificación (BGR8, RGB8, etc.), ancho, alto, y una cabecera (header) con el timestamp (cuándo se tomó la imagen) y el frame_id (de qué cámara viene).
  - std_msgs.msg.Float32MultiArray: Es un tipo de mensaje simple de ROS 2 que contiene una lista de números flotantes (como tus datos de *offset* y distancia).
- **CvBridge:**
  - from cv_bridge import CvBridge: Esta es una biblioteca auxiliar muy útil.

CvBridge actúa como un **puente** entre las imágenes en formato de OpenCV (que son matrices de NumPy) y los mensajes de imagen de ROS 2 (sensor_msgs.msg.Image).
    - self.bridge.cv2_to_imgmsg(...): Convierte una imagen de OpenCV a un mensaje ROS Image.
    - self.bridge.imgmsg_to_cv2(...): Convierte un mensaje ROS Image a una imagen de OpenCV.
- **QoSProfile (Calidad de Servicio):**
    - QoSProfile define cómo se manejan los mensajes en los topics. Es como decirle al cartero si una carta es "urgente" o si "no importa si pierde alguna".
    - ReliabilityPolicy.BEST_EFFORT: "Haz tu mejor esfuerzo". Los mensajes se envían lo más rápido posible, pero no se garantiza que lleguen todos (útil para vídeo, donde el último frame es el más importante y no quieres retrasos por retransmisiones).
    - ReliabilityPolicy.RELIABLE: "Fiable". Se garantiza que todos los mensajes lleguen al destino (útil para comandos, datos críticos, donde no quieres perder información).
    - HistoryPolicy.KEEP_LAST: Solo guarda el último mensaje.
    - depth: Cuántos mensajes quieres guardar en el historial.

7. Flujo de Datos entre tus Nodos (Un Ejemplo Sencillo)

Para que veas cómo se conectan las piezas:

1. **NodoCamaraTello (El Ojo y el Controlador del Tello):**
    - Conecta con el Tello.
    - Captura el frame de vídeo del Tello.
    - **Publica** ese frame como Image en /tello/imagen_raw.
    - **Se suscribe** a /tello/comandos_velocidad para recibir órdenes y enviarlas al Tello.
2. **NodoProcesadorImagen (El Cerebro Visual):**
    - **Se suscribe** a /tello/imagen_raw para recibir los frames del Tello.
    - Procesa la imagen (detección de color azul, contornos, cálculo de distancia).
    - **Publica** la imagen procesada (con los dibujos de la detección) en /tello/imagen_procesada.
    - **Publica** los datos numéricos de la detección (offset_x, offset_y, distancia) como Float32MultiArray en /tello/datos_deteccion.
3. **NodoControlLogica (El Cerebro de Decisión):**
    - **Se suscribe** a /tello/datos_deteccion para recibir la información procesada.
    - Usa esos datos (offset, distancia) para decidir cómo debe moverse el Tello (ej.

ir a la izquierda, subir, avanzar).
    - **Publica** los comandos de velocidad (valores lr, fb, ud, yv) como Float32MultiArray en /tello/comandos_velocidad.

¡Y el ciclo se repite! La información fluye de un nodo a otro, permitiendo que tu dron Tello navegue de forma autónoma.

Espero que esta guía te aclare todas tus dudas y te sirva como referencia rápida. ¡Mucho ánimo con tu proyecto de robótica, vas por muy buen camino!