Python App Documentation for Drone Interaction

Overview

This documentation outlines the design and functionality of a Python application developed by a third-party to interact with drones. The app enables the drone to perform specific tasks by defining and executing functions tailored for specialized missions, such as inspecting cracks in a pillar at an inspection site. The drone performs these tasks using input parameters defined in a list and returns output to the user. Successfully executed functions can be stored as reusable capabilities in a **Behavior Tree** for future missions.

Key Features

1. **Dynamic Function Definitions**: Developers can define drone actions by specifying parameters in a list (e.g., `image`, `location`, `height`, and `angle`) along with their respective data types.
2. **Drone Execution**: The drone reads the input, executes the defined functions, and collects the required data.
3. **Output Validation**: After execution, the app returns `True` or `False` to indicate the success of the task.
4. **Reusable Capabilities**: Successfully executed functions can be stored as reusable behaviors in the app's **Behavior Tree** under the capabilities module, enabling future use.

System Workflow

1. Input Definition

The developer defines the task by providing input parameters in a list:

- **Parameter List Format**: `[parameter_name, data_type]`

Example:

```python
task_parameters = [ {"name": "image", "type": "str"}, {"name": "location",
"type": "tuple"}, {"name": "height", "type": "float"}, {"name": "angle",
"type": "float"} ]
```

2. Drone Execution

- The Python app sends the defined parameters to the drone.
- The drone navigates based on `location`, `height`, and `angle`.
- It collects and processes the data (e.g., capturing an image of the crack).
- The drone returns the result of the task as `True` (success) or `False` (failure).

3. Storing Capabilities

- After successful task execution, the function can be added as a reusable capability.
- The developer calls `capabilities.add(<function_name>)` to save the function in the **Behavior Tree**.
- This allows seamless integration and execution in future missions.

Code Example

Defining a Task

Python

```
# Define the task parameters

task_parameters = [ {"name": "image", "type": "str"}, {"name": "location",
"type": "tuple"},  # Example: (latitude, longitude){"name": "height", "type":
"float"},  # Drone height in meters {"name": "angle", "type": "float"}  #
Camera angle in degrees ]

# Function to navigate and inspect cracks

def navigate_crack(parameters):

""" Navigate to the location, inspect cracks, and return the status. Args:
parameters (list): List of parameters with data types. Returns: bool: True if
the task was successful, False otherwise. """

try:
       # Extracting parameters
       image = parameters.get("image")
       location = parameters.get("location")
       height = parameters.get("height")
       angle = parameters.get("angle")

       # Simulate drone operation
       print(f"Navigating to location: {location} at height: {height} meters.")
       print(f"Capturing image with angle: {angle}°")

       # Simulated result
       success = True # Replace with actual drone API call and logic

       return success except Exception as e:
       print(f"Error during task execution: {e}")
       return False
```

Adding to Behavior Tree

```python
# Adding the function as a capability def add_to_capabilities(func_name): """
Adds a successfully executed function to the Behavior Tree as a reusable
capability. Args: func_name (str): Name of the function to add. """ try: #
Example Behavior Tree addition capabilities.add(func_name) print(f"Capability
'{func_name}' added successfully.") except Exception as e: print(f"Error adding
capability: {e}")
```

Main Program

```python
if __name__ == "__main__":
# Execute the task
result = navigate_crack({
 "image": "crack_image.jpg",
 "location": (12.971598, 77.594566),  # Example: Bangalore coordinates
"height": 10.0,
 "angle": 45.0
})

# Check the result and add capability if successful if result: print("Task
completed successfully!") add_to_capabilities("navigate_crack") else:
print("Task failed. Please review the parameters or environment.")
```

Integration with Behavior Tree

The **Behavior Tree** serves as a repository and decision-making framework for drone actions:

- **Capability Registration**: Functions can be added to the tree with:

```python
capabilities.add("function_name")
```

- **Execution in Future Missions**: Once stored, these capabilities can be reused in other scenarios without redefinition.

Error Handling

- **Input Validation**: Ensure all parameters are correctly defined and match the expected data types.
- **Execution Monitoring**: Handle errors like signal loss or GPS failure with exception handling.
- **Output Assurance**: Validate the drone's output before confirming task completion.

Future Enhancements

- **Dynamic Parameter Updates**: Enable real-time parameter modifications during task execution.
- **Logging and Analytics**: Provide detailed logs of each task for debugging and performance analysis.
- **AI Integration**: Enhance decision-making with AI for advanced inspections or obstacle avoidance.

This documentation provides a comprehensive guide for integrating drones with the Python app, enabling efficient task execution and reusable behavior storage.