

# Natural Language Command and Control for UAVs

## Assignment 1

Advika Goel: 250076

### Ques 1)

**Explain what happens if you define a function like this: "def add\_item(item, box=[]):".**

**Why does the list "box" persist data between function calls if you don't provide a second argument?**

Ans:

The issue with defining a function like `def add_item(item, box=[]):` stems from how Python handles function definitions. In Python, default argument values are evaluated only once when the function is defined, not every time the function is called. Because lists are mutable objects, the list created at definition stays in memory. If you modify this list (e.g., by appending an item), the changes persist across future function calls that rely on the default argument.

To fix this, the best practice is to set the default value to `None`. Inside the function, you can check if the argument is `None` and create a new list then. This ensures a fresh list is created for every call where the argument is omitted.

### Ques 2) `__str__` vs `__repr__`

**Both methods return string representations of an object. What is the strict technical difference in their intended audience?**

**Which one is used as a fallback if the other is missing?**

Ans:

The strict technical difference between these two methods lies in their intended audience.

- `__str__` is designed to provide a readable, user-friendly string representation of an object. It is what users see when they print an object or convert it to a string.
- `__repr__` is intended for developers. It should provide an unambiguous representation of the object, ideally a valid string of code that could be used to recreate the object (e.g., `ClassName(arg1, arg2)`).

If a class does not implement `__str__`, Python will use `__repr__` as a fallback.

### Ques 3)

## **Class Variables vs. Instance Variables**

**Explain the memory difference between a variable defined inside "\_\_init\_\_" (using self.var) versus a variable defined directly under the "class Name:" header.**

**If you change a Class Variable using "ClassName.var = new\_value", what happens to existing instances? What happens if you try to change it via "instance.var = new\_value"?**

Ans: There is a distinct difference in how memory is allocated for these variables. A variable defined inside \_\_init\_\_ using self.var is an instance variable, meaning every object (instance) created gets its own separate copy of that variable in memory. A variable defined directly under the class header is a class variable, which is shared across all instances of that class. There is only one copy of a class variable in memory, regardless of how many instances exist.

**Behavior When Modified:**

- If we modify a class variable using the class name (e.g., ClassName.var = new\_value), the change is reflected across all instances immediately because they are all referencing the same shared data.
- If we try to change it through a specific instance (e.g., instance.var = new\_value), Python creates a new instance variable with that name for that specific object. This "shadows" or hides the class variable for that instance only, leaving the shared class variable and all other instances unchanged.