Class or Static Variables in Python

In Python, a static variable is a variable that is shared among all instances of a class, rather than being unique to each instance. It is also sometimes referred to as a class variable because it belongs to the class itself rather than any particular instance of the class.

Static variables are defined inside the class definition, but outside of any method definitions. They are typically initialized with a value, just like an instance variable, but they can be accessed and modified through the class itself, rather than through an instance.

Features of Static Variables

- **Memory Efficiency**: Static variables are allocated memory once when the object for the class is created for the first time.
- **Class Scope**: Static variables are created outside methods but inside the class.
- **Access Through Class**: Static variables can be accessed through the class but not directly through an instance.
- **Consistent Behavior**: The behavior of static variables doesn't change for every object.

The Python approach is simple; it doesn't require a static keyword.

Note: *All variables which are assigned a value in* the *class declaration are class variables. And variables* that *are assigned values inside methods are instance variables.*

Example of Static Variables in Python

```
# Python program to show that the variables with a value
# assigned in class declaration, are class variables


# Class for Computer Science Student
class CSStudent:
    stream = 'cse'                # Class Variable
    def __init__(self,name,roll):
        self.name = name          # Instance Variable
        self.roll = roll          # Instance Variable


# Objects of CSStudent class
a = CSStudent('Geek', 1)
b = CSStudent('Nerd', 2)


print(a.stream)  # prints "cse"
print(b.stream)  # prints "cse"
print(a.name)    # prints "Geek"
```

```
print(b.name)    # prints "Nerd"

print(a.roll)    # prints "1"

print(b.roll)    # prints "2"


# Class variables can be accessed using class

# name also

print(CSStudent.stream) # prints "cse"


# Now if we change the stream for just 'a', it won't be changed for 'b'

a.stream = 'ece'

print(a.stream) # prints 'ece'

print(b.stream) # prints 'cse'


# To change the stream for all instances of the class, we can change it

# directly from the class

CSStudent.stream = 'mech'


print(a.stream) # prints 'ece'

print(b.stream) # prints 'mech'
```

Output

```
cse
cse
Geek
Nerd
1
2
cse
ece
cse
ece
mech
```

This example shows how **class variables** (or static variables) are shared across all instances. However, when an instance modifies the class variable, it creates its own copy, which leads to a different behavior.

Key Differences Between Class Variables in Python and Static Variables in Java/C++

While class variables in Python and static variables in Java/C++ serve a similar purpose of being shared across all instances, they behave differently when modified through an instance:

- **Java/C++ Behavior**: When you modify a static variable in Java or C++, the change is reflected across all instances of the class, and they all remain synchronized with the static variable's value.
- **Python Behavior**: In Python, if you modify a class variable through an instance, a new instance variable is created. This separates the modified value from the original class variable, which remains unchanged for other instances.

Example to Demonstrate the Difference

```
class MyClass:
    class_var = 'original'  # Class variable


# Creating two instances
obj1 = MyClass()
obj2 = MyClass()


# Changing class_var for obj1 only
obj1.class_var = 'modified'


# Outputs
print(obj1.class_var)  # Output: 'modified'
print(obj2.class_var)  # Output: 'original'
print(MyClass.class_var)  # Output: 'original'
```

Output

```
modified
original
original
```

Explanation: obj1 has created its own version of class_var, separating it from the class-level class_var. As a result, the class variable remains unchanged for other instances.

Instance Variables vs Class Variables in Python

- **Instance Variables**: Variables that are unique to each object. They are created inside methods (typically __init__()).
- **Class Variables (Static Variables)**: Variables shared among all instances of a class. They are defined within the class, outside any methods.

Modified Example to Show Behavior

```
class MyClass:
    static_var = 0  # Class variable


    def __init__(self):
        MyClass.static_var += 1  # Modify through class name
        self.instance_var = MyClass.static_var  # Instance variable


obj1 = MyClass()
print("obj1.instance_var:", obj1.instance_var)  # Output: 1
```

```
obj2 = MyClass()
print("obj2.instance_var:", obj2.instance_var)  # Output: 2

# Access class variable directly
print("MyClass.static_var:", MyClass.static_var)  # Output: 2

# Modify class variable using obj1 (this creates a new instance variable)
obj1.static_var = 10
print("obj1.static_var:", obj1.static_var)  # Output: 10 (instance variable now)
print("MyClass.static_var:", MyClass.static_var)  # Output: 2 (unchanged)
print("obj2.static_var:", obj2.static_var)  # Output: 2 (unchanged)
```

Output

```
obj1.instance_var: 1
obj2.instance_var: 2
MyClass.static_var: 2
obj1.static_var: 10
MyClass.static_var: 2
obj2.static_var: 2
```

Best Practices for Using Class Variables

- **Use Class Variables for Shared Data**: If you want all instances to share a particular value or state, use class variables.
- **Modify Class Variables via Class Name**: Always modify class variables using the class name (ClassName.variable) to avoid unintentionally creating an instance-level copy.
- **Avoid Modifying Class Variables from Instances**: Modifying class variables through instances can lead to confusion, as it creates instance-level variables.
- **Be Careful in Multithreaded Environments**: In multithreaded programs, unsynchronized class variables can cause race conditions.

Advantages:

- **Memory efficiency**: Since static variables are shared among all instances of a class, they can save memory by avoiding the need to create multiple copies of the same data.
- **Shared state:** Static variables can provide a way to maintain shared state across all instances of a class, allowing all instances to access and modify the same data.
- **Easy to access:** Static variables can be accessed using the class name itself, without needing an instance of the class. This can make it more convenient to access and modify the data stored in a static variable.
- **Initialization**: Static variables can be initialized when the class is defined, making it easy to ensure that the variable has a valid starting value.
- **Readability**: Static variables can improve the readability of the code, as they clearly indicate that the data stored in the variable is shared among all instances of the class.

Disadvantages:

- **Inflexibility**: Static variables can be inflexible, as their values are shared across all instances of the class, making it difficult to have different values for different instances.
- **Hidden dependencies**: Static variables can create hidden dependencies between different parts of the code, making it difficult to understand and modify the code.
- **Thread safety**: Static variables can be problematic in a multithreaded environment, as they can introduce race conditions and synchronization issues if not properly synchronized.
- **Namespace pollution**: Static variables can add to the namespace of the class, potentially causing name conflicts and making it harder to maintain the code.
- **Testing:** Static variables can make it more difficult to write effective unit tests, as the state of the static variable may affect the behavior of the class and its methods.

Overall, static variables can be a useful tool in Python programming, but they should be used with care and attention to potential downsides, such as inflexibility, hidden dependencies, and thread safety concerns.

Conclusion

Class variables, or static variables, in Python can be a powerful tool when used correctly. They allow you to share data across all instances of a class, but they need to be handled carefully to avoid hidden dependencies or unexpected behavior. Always remember to modify them through the class itself to prevent creating instance-specific variables unintentionally.