Programming Problem 01: Object-Oriented Design and Implementation

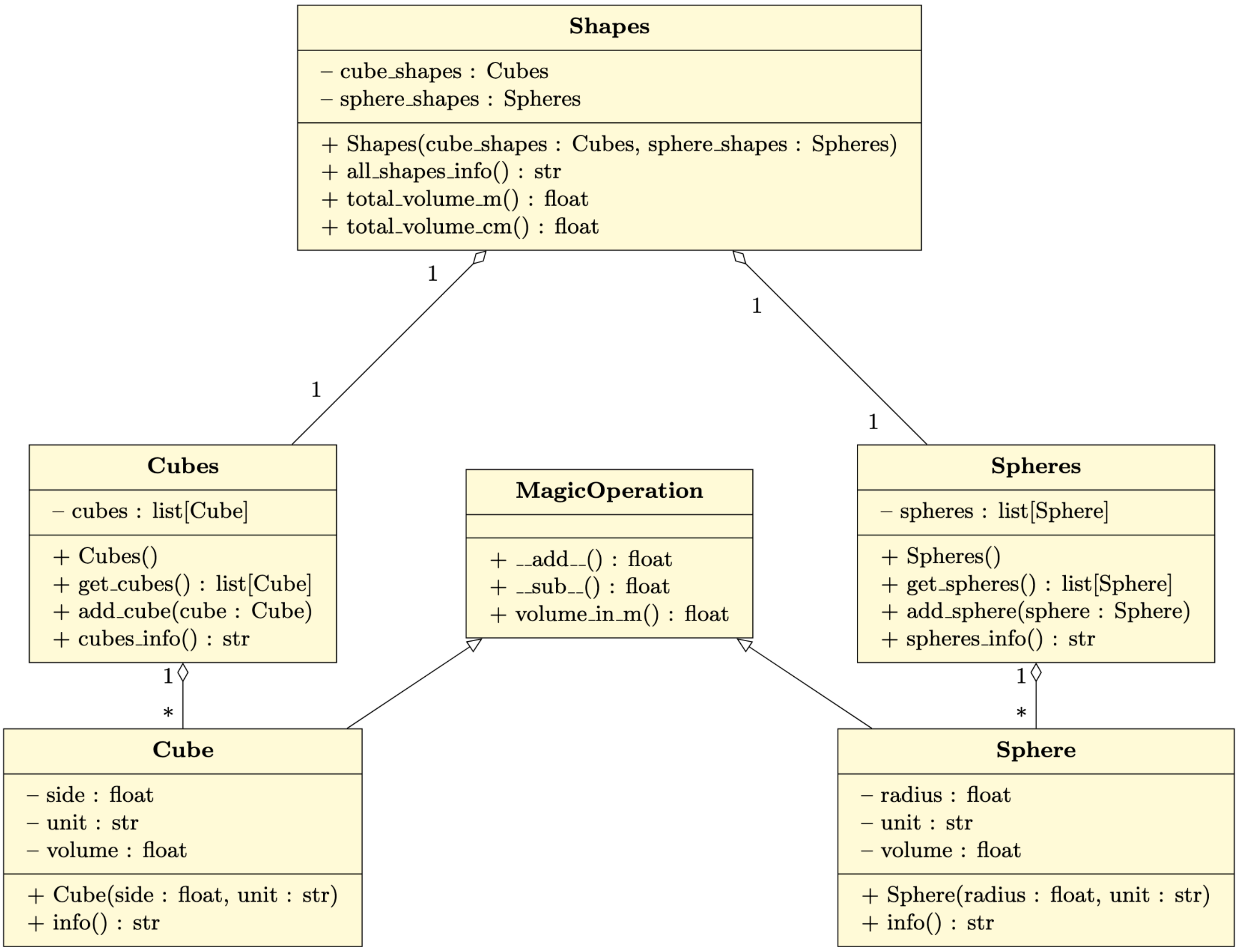
Submission:

For this problem, at the end, you must submit a file named **problem01_classes.py**, which contains your class implementations. **Submissions with an incorrect filename, incomplete content, or late submissions will not be permitted for resubmission in any circumstances.** **Please manage your time properly for the submission process.**

Other files are only for demonstrating class usage and are not part of the submission. Keep in mind that actual usage may differ from the provided examples.

Task:

The UML diagram below represents a class structure for handling geometric shapes (Cubes and Spheres). Implement the following classes in Python according to the UML diagram.



Requirements:

General requirements:

1. **The examples and expected output are part of the question. You must implement classes that match the example usages to provide exactly the same output (including every character, symbol, spacing, and character case).**
2. **The computer will be used to grade this problem. There will be no score for any program that cannot be run.**
3. The program must execute the given example and produce the expected output, including usage beyond the test cases.
4. Allow only Python standard library; no other libraries are allowed.
5. The implementation must strictly follow the UML class diagram. All classes with provided details must be implemented.
6. Additional implementation details beyond the UML diagram are allowed.
7. The mathematical constant π (Pi) must be used from math.pi for all calculations.
8. All numerical values must be stored with full precision in variables and displayed with two decimal places using formatted output.
9. The other example files, which are not mentioned in detail below, are for helping you validate your implementation.

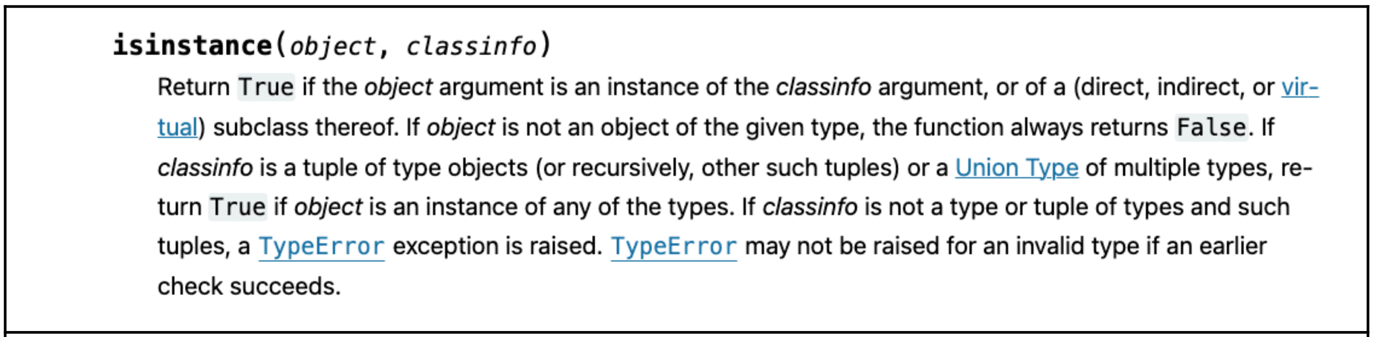
Example ex01.py:

1. The valid values for side and radius in Cube and Sphere are zero or positive values. The valid unit must be either 'cm' or 'm' (in lowercase only). For invalid values, the constructor should raise ValueError "The values are invalid"
2. Some methods return string data type, meaning that formatting and string construction without printing should occur within the method itself.
3. cubes_info() should be given the title "Cubes:", followed by shape numbering and cube details (reuse info()). If there are no cubes, the cube details should be "No cubes available"
4. spheres_info() should be given the title "Spheres:", followed by shape numbering and sphere details (reuse info()). If there are no spheres, the sphere details should be "No spheres available"
5. all_shapes_info() should reuse the cubes_info() and spheres_info() to create the output. The output should contain the total volume in cubic meters (m3) and cubic centimeters (cm3).

Example ex02.py:

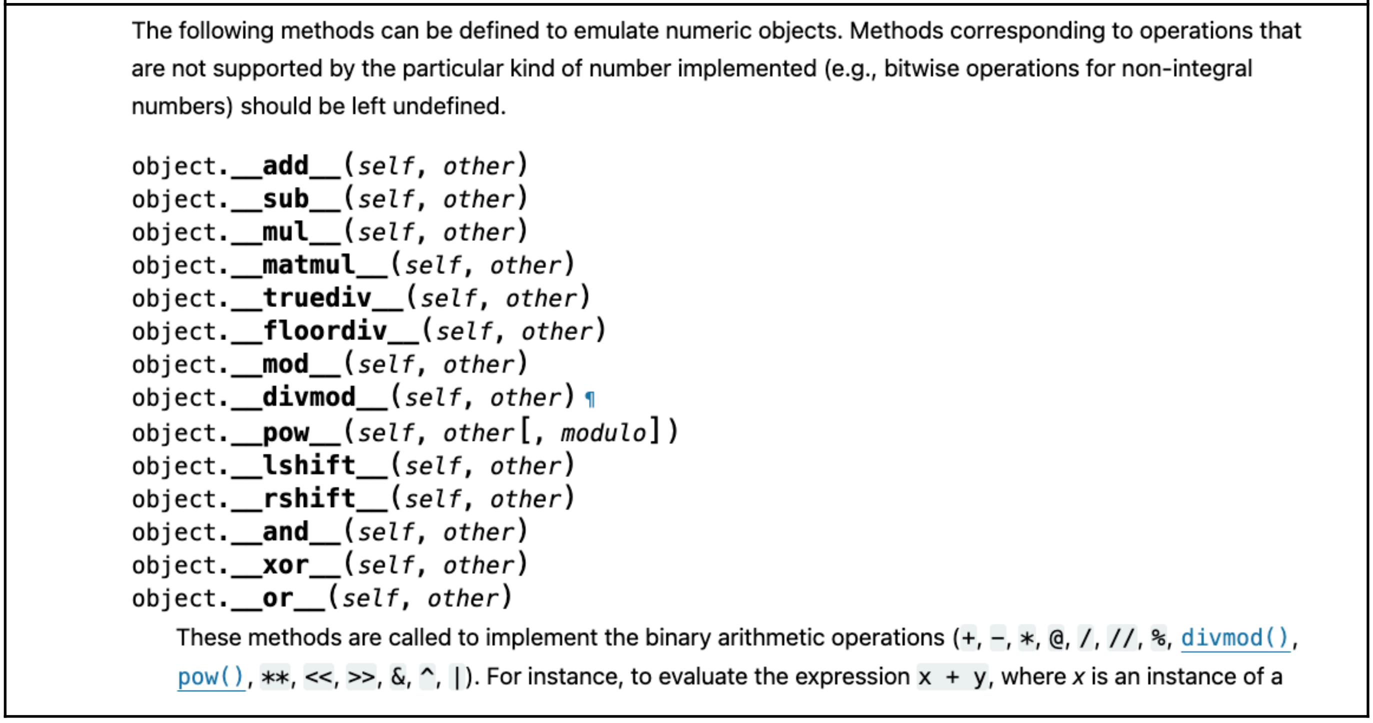
1. Cube and Sphere must inherit from the MagicOperation class.
2. Cube and Sphere must not contain any magic methods. They must use the __add__, __sub__, and volume_in_m() methods from the parent.
3. The __add__ and __sub__ methods must only accept instances of Cube or Sphere, validated using isinstance(), and raise ValueError "Type does not match" if invalid.
4. The __add__ method must return a float representing the sum of the volumes of itself and other instances in a cubic meter (m3).
5. The __sub__ method must return a float representing the difference between the volumes of itself and other instances in a cubic meter (m3).

Hint from Python document:



isinstance(*object*, *classinfo*)

Return `True` if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect, or virtual) subclass thereof. If *object* is not an object of the given type, the function always returns `False`. If *classinfo* is a tuple of type objects (or recursively, other such tuples) or a Union Type of multiple types, return `True` if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a `TypeError` exception is raised. `TypeError` may not be raised for an invalid type if an earlier check succeeds.



The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(self, other) ¶
object.__pow__(self, other[, modulo])
object.__lshift__(self, other)
object.__rshift__(self, other)
object.__and__(self, other)
object.__xor__(self, other)
object.__or__(self, other)
```

These methods are called to implement the binary arithmetic operations (`+`, `-`, `*`, `@`, `/`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). For instance, to evaluate the expression `x + y`, where *x* is an instance of a