```
"""
Encapsulation:
Process of Binding attributes(variables) and methods(functions)
as a single entity.

Access Specifiers:
Private:
The attributes and methods declared as private can only be accessed
within the same class.
They cannot be accessed from child classes and from outside.
We use __ at the beginning of the attribute or method to declare
it as private.

Protected:
The Attributes and methods declared as protected can be accessed
within the same class and the classes which inherit that class
(child classes).
Although Python doesn't strictly enforce or restrict the access,
We shouldn't try to access them from outside.
We use _ at the beginning of an attribute or method to declare it
as public.

Public:
The Attributes and methods declared as public can be accessed from
anywhere. (Within the same class, its sub-classes and from outside).

Uses of Encapsulation:
1. Data Security: Protects data from unauthorized or accidental modification.
2. Code Organization: Promotes a clear and organized structure by grouping related
data and functionality.
3. Modularity and Reusability: Encapsulated units (classes) can be easily reused
and integrated into different parts of a program or other projects.
4. Maintainability: Changes to the internal implementation of a class are less
likely to impact external code, as long as the public interface remains consistent.
"""
class A:
    def __init__(self, value1, value2, value3):
        self.__private_member = value1
        self._protected_member = value2
        self.public_member = value3

    def __private_method(self):
        print("I am Private Method inside Class A")

    def _protected_method(self):
        print("I am Protected Method inside Class A")

    def public_method(self):
        print("I am Public Method inside Class A")


class B(A):
    def __init__(self, value1, value2, value3):
        super().__init__(value1, value2, value3)

    def method(self):
        # print(self.__private_member)
        # super().__private_method()
        print(self._protected_member)
```

```
        self._protected_method()
        print(self.public_member)
        self.public_method()
        pass

b = B(10, 20, 30)
b.method()
# print(b.__private_member)
# print(b.__private_method())
# print(b._protected_member)
# b._protected_method()
print(b.public_member)
b.public_method()


class Foo:
    def __init__(self):
        self.a = 10

f = Foo()
f.new_attr = "Value"
print(f.new_attr)


class Foo:
    def __init__(self):
        self.__a = 10
        self.__b = 20


class Bar(Foo):
    def __init__(self):
        super().__init__()
        self.c = 30
        self.d = 40


b = Bar()
# print(b.__a)


class BankAccount:
    def __init__(self, bank_balance):
        self.__bank_balance = bank_balance

    def set_bank_balance(self, new_balance):
        self.__bank_balance = new_balance

    def fetch_bank_balance(self):
        return self.__bank_balance

    def deposit(self, deposit_amount):
        new_balance = self.__bank_balance + deposit_amount
        self.set_bank_balance(new_balance)

    def withdraw(self, withdrawl_amount):
        new_balance = self.__bank_balance - withdrawl_amount
        self.set_bank_balance(new_balance)
```

```
bank_account = BankAccount(1000)
bank_account.deposit(500)
print(bank_account.fetch_bank_balance())


"""
Name Mangling:
Name mangling in Python is a mechanism that automatically renames
identifiers (like attributes or methods) within a class
when they are prefixed with two leading underscores
(e.g., __attribute_name).
This renaming process is primarily designed to prevent
name clashes in subclasses, especially when a subclass might
inadvertently override an attribute or method from its parent class
that shares the same name.

How it works:
When an attribute or method name within a class starts with two
underscores and does not end with two underscores,
Python automatically transforms its name.
The transformation involves adding a single underscore
followed by the class name, and then the original identifier.
Example:
class MyClass:
    def __init__(self):
        self.__private_attribute = "I am private"

    def __private_method(self):
        return "This is a private method"

# Accessing the mangled name
obj = MyClass()
print(obj._MyClass__private_attribute)
print(obj._MyClass__private_method())

Purpose:
While name mangling can create a perception of "private" attributes,
it's important to understand that it does not enforce true privacy
like in languages such as Java or C++. All attributes in Python are
ultimately accessible. The primary purpose of name mangling is to
prevent accidental name collisions when working with inheritance,
ensuring that a subclass doesn't unintentionally overwrite an internal
attribute or method of a superclass.

Important Note:
Name mangling does not occur if an identifier starts and
ends with two underscores (e.g., __init__, __str__, __repr__, __eq__).
These are typically special methods (often called "dunder methods")
that have specific meanings in Python and are not intended for
mangling.
"""

class Foo:
    def __init__(self):
        self.__private_attribute = "Private Value"

    def __private_method(self):
        print("Inside Private Method...")
```

```
f = Foo()
#print(f.__private_attribute)
print(f._Foo__private_attribute)
```