Python - public, private and protected

Classical object-oriented languages, such as C++ and Java, control the access to class resources by public, private and protected keywords. Private members of a class are denied access from the environment outside the class. They can be handled only from within the class.

Public members (generally methods declared in a class) are accessible from outside the class. The object of the same class is required to invoke a public method. This arrangement of private instance variables and public methods ensures the principle of data encapsulation.

Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class.

Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with single or double underscore to emulate the behaviour of protected and private access specifiers.

All members in a Python class are **public** by default. Any member can be accessed from outside the class environment.

Example: Public Attributes

```
class employee:
    def __init__(self, name, sal):
        self.name=name
        self.salary=sal
```

You can access employee class's attributes and also modify their values, as shown below.

```
>>> e1=employee("Kiran",10000)
>>> e1.salary
10000
>>> e1.salary=20000
>>> e1.salary
20000
```

Python's convention to make an instance variable **protected** is to add a prefix _ (single underscore) to it. This effectively prevents it to be accessed, unless it is from within a sub-class.

Example: Protected Attributes

```
class employee:
    def __init__(self, name, sal):
        self._name=name  # protected attribute
        self._salary=sal # protected attribute
```

In fact, this doesn't prevent instance variables from accessing or modifyingthe instance. You can still perform the following operations:

```
>>> e1=employee("Swati", 10000)
>>> e1._salary
10000
>>> e1._salary=20000
>>> e1._salary
20000
```

Hence, the responsible programmer would refrain from accessing and modifying instance variables prefixed with _ from outside its class.

Similarly, a double underscore __ prefixed to a variable makes it **private**. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

Example: Private Attributes

```
class employee:
    def __init__(self, name, sal):
        self.__name=name  # private attribute
        self.__salary=sal # private attribute
>>> e1=employee("Bill",10000)
>>> e1.__salary
AttributeError: 'employee' object has no attribute '__salary'
```

Python performs name mangling of private variables. Every member with double underscore will be changed to _object._class__variable. If so required, it can still be accessed from outside the class, but the practice should be refrained.

```
>>> e1=employee("Bill",10000)
>>> e1._employee__salary
10000
>>> e1._employee__salary=20000
>>> e1._employee__salary
20000
```