Advanced Assignment 5

Q1. What is the meaning of multiple inheritance?

Answer: Multiple inheritance occurs when a class inherits from more than one superclass; it's useful for mixing together multiple packages of class-based code. The left-to-right order in class statement headers determines the order of attribute searches.

```
class A:
    def a(self):
        print('called A.a()')

class B:
    def b(self):
        print('called B.b()')

class C(A, B): # Multiple inheritance
    pass

c = C()
c.a() # called A.a()
c.b() # called B.b()
```

Q2. What is the concept of delegation?

Answer: Delegation involves wrapping an object in a proxy class, which adds extra behaviour and passes other operations to the wrapped object. The proxy retains the interface of the wrapped object.

```
class UpperOut:
    def __init__(self, outfile):
        self.outfile = outfile
    def write(self, s:str):
        return self.outfile.write(s.upper())
with open('output.txt', 'w') as f:
    upper_out = UpperOut(f)
    upper_out.write('hello world')
```

The UpperOut class is defined with an __init__ method and a write method. The __init__ method takes one argument, outfile, which is expected to be a file-like object with a write method. The outfile object is stored as an instance variable `outfile`.

The write method of the UpperOut class takes one argument, s, which is expected to be a string. The method calls the write method of the outfile object, but modifies its behaviour by converting the data to uppercase before writing it.

Q3. What is the concept of composition?

Answer:

- Composition is an object-oriented programming technique where one class contains another class as a component. This helps to build complex objects from simpler ones by combining their behaviours.
- The contained object is usually accessed through a member variable of the containing object, which then delegates some of its behaviour to the contained object.
- Composition can be used when various components of a problem reflect a `*has-a*` relationship. For example, electric car has a battery.

```
class Battery:
    def __init__(self, capacity):
        self.capacity = capacity

    def get_capacity(self):
        return self.capacity

class ElectricCar:
    def __init__(self, battery_capacity):
        self.battery = Battery(battery_capacity)

    def get_battery_capacity(self):
        return self.battery.get_capacity()
```

Q4. What are bound methods and how do we use them?

Answer:A bound method is a method that is dependent on the instance of the class. It passes the instance as the first argument which is used to access the variables and functions. In Python 3 and newer versions of python, all functions in a class are by default bound methods.

```
class MyClass:
    def __init__(self):
        self.x = 10

    def my_method(self):
        print(self.x)

my_instance = MyClass()
my_instance.my_method() # prints 10
```

In this example, my_method is a bound method because it is called on an instance of MyClass (my_instance) and has access to the instance variables (self.x).
When we call my_instance.my_method(), the instance my_instance is automatically passed as the first argument to the method (self).

Q5. What is the purpose of pseudo-private attributes?

Answer:

- Pseudo-private attributes are used to prevent accidental access to an attribute from outside the class by localising names to the enclosing class.
- Pseudo-private attributes are created by prefixing the attribute name with two underscores (__). This includes both class attributes like methods defined inside the class, and self instance attributes assigned inside the class.
- This causes the attribute name to be “mangled” by having the class name prefixed to it with an additional underscore which makes it harder to accidentally access or modify the attribute from outside the class. Such names are expanded to include the class name, which makes them unique.

```
class MyClass:
    def __init__(self):
        self.__x = 10
    def print_x(self):
        print(self.__x)

my_instance = MyClass()
my_instance.print_x() # prints 10
print(my_instance.__x) # raises an AttributeError
```